TU-Wien

Institut E114

Prof. DDr. F. Rattay

Computersimulation für Elektrotechniker

Die Programmiersprachen C und C++

1. Einführung 4

1.1. „Hochsprachen“ und maschinennahe Sprachen 4
1.2. Spezialisierung bestimmter Programmiersprachen 5
1.3. Übersicht über einige Programmiersprachen 7
1.4. Die verwendete IDE 8
1.4.1. Starten der IDE und Anlegen eines Projektes 8
1.4.2. Eingabe des Quelltextes 10
1.4.3. Übersetzen und Linken 10
1.4.4. Ausführen und Debugging des Programms 11

2. Grundbegriffe und Sprachstrukturen 14

2.1. Bezeichner 14
2.2. Schlüsselwörter 14
2.3. Aufbau eines C-Programms 15
2.4. Einbinden von Header-Dateien 15
2.5. Die Funktion main 15

3. Variablen - Einfache Datentypen 17

3.1. Variablennamen 17
3.2. Datentypen 18
3.3. Deklaration 19

4. Kontrollstrukturen 21

5. Zeiger 26

5.1. Speicherverwaltung mit Zeigern 27

6. Funktionen 29

7. Benutzerdefinierte Datenstrukturen, Structs, Unions 31

7.1. "Structs" 31
7.1.1. Zugriff auf Komponenten 34
7.2. Neue Typnamen erzeugen mit "typedef" 35
7.3. "Unions" 40

8. Literatur zu der Programmiersprache C 43

9. Klassen und Objekte 44

9.1. Beispiel: Bankkonto 44
9.2. Eine Klasse in C++ 45
9.3. Erzeugen von Objekten 49
9.4. Konstruktoren und Destruktoren 51
9.5. Dynamische Objekte 52

10. Member-Funktionen, Zugriffsrechte und andere Konstrukte 56

10.1. Friends 65
10.2. Referenzen 68
10.3. Konstanten 71

11. Überladen von Operatoren und Funktionen 74

11.1. Überladen von Funktionen 74
11.2. Das Überladen von Operatoren: 76

12. Streams und File I/O 82

12.1. Manipulatoren 82
12.2. Format-Funktionen 83
12.3. Status-Abfragen 86
12.4. I/O-Operatoren auf eigenen Datentypen 88
12.5. Ein-/Ausgabe mit Dateien 90

13. Vererbung 93

14. Literatur zur Programmiersprache C++ 97

Einführung

Um dem Leser einen Überblick über verschiedene, Programmiersprachen, Konzepte und Entwicklungsumgebungen zu geben, sollen im folgenden die wesentlichen Eigenschaften der in diesem Skript behandelten Programmiersprache „C“ im Vergleich zu anderen Sprachen dargestellt werden.

„Hochsprachen“ und maschinennahe Sprachen

Prinzipiell benutzt man Programmiersprachen um dem Computer eine genaue Beschreibung einer von ihm zu erledigenden Aufgabe zu geben. Da man diese Beschreibung nicht einfach umgangssprachlich formulieren kann, ist eine Kommunikationsmethode erforderlich, die sowohl vom Rechner als auch vom Menschen „verstanden“ wird. Von entscheidender Bedeutung ist die Frage, wie weit der Programmierer sich an den Rechner annähern muß, um eine Programm entwerfen zu können, bzw. wie weit ihm der Rechner, oder besser gesagt die verwendete Programmiersprache, entgegenkommt. Man spricht in diesem Zusammenhang von sogenannten „Hochsprachen“, die dem Programmierer ein relativ hohes Abstraktionsniveau bieten und im Gegensatz dazu von „Assembler“ oder „Maschinensprachen“. Um ein Programm in Assembler zu programmieren sind genaueste Kenntnisse über den internen Aufbau des Rechners notwendig. Der Programmierer muß z.B. wissen, wie die CPU des Rechners aufgebaut ist, aus welchen Registern sie besteht und an welchen Adressen im Speicher ein bestimmtes Betriebssystem-Unterprogramm (s. Kapitel 0, Kapitel 0) anfängt. Assembler bietet aber immerhin (im Unterschied zur reinen Maschinensprache) die Möglichkeit Befehle mit einigermaßen verständlichen Kürzeln darzustellen, z.B. wird ein Sprungbefehl, um an eine andere Stelle im Programm zu gelangen, oft durch „JMP“ (für JuMP) abgekürzt.
Die reine Maschinensprache, die hauptsächlich in der Anfangszeit der Programmierung benutzt wurde, bietet nicht einmal diesen Komfort, sondern codiert alle nötigen Anweisungen direkt im binären Format, der einzigen Sprache, die der Rechner direkt versteht. Trotz des hohen Aufwandes und der großen Fehleranfälligkeit, die Assembler-Programme mit sich bringen, haben sie auch heute noch ihre Berechtigung, da sie nämlich in der Regel sehr schnell und sehr kompakt sind. Aus diesen Gründen wird Assembler oft in systemnahen oder sehr geschwindigkeitsabhängigen Bereichen, wie z.B. der Programmierung von Treibersoftware, verwendet.
Ein sehr viel höheres Abstraktionsniveau bieten die „Hochsprachen“, wie etwa BASIC, Pascal, COBOL, Smalltalk usw. sowie die visuellen Entwicklungssysteme neuerer Bauart, wie etwa Visual BASIC, Delphi oder Powerbuilder. Bei Verwendung einer solchen „höheren“ Programmiersprache sind genaue interne Kenntnisse des Rechners nicht mehr notwendig. Es werden Befehle und Anweisungen angeboten, die auf einer höheren Abstraktionsstufe liegen. So kann man z.B. Befehle zur einfachen Ausgabe von Text auf dem Bildschirm verwenden, wie etwa writeln ("Hello World"); in Pascal, die relativ einfach zu verstehen sind. Die gleiche Anweisung in Assembler besteht aus einer großen Anzahl kryptischer Befehle und Zahlen im Hexadezimalformat [1]. Um einen Eindruck über die unterschiedlichen Abstraktionsniveaus zu erhalten, ist im folgenden ein einfaches Beispielprogramm einmal in C und einmal in Assembler dargestellt:

C++ Code:

```
#include "stdio.h"
main()
{
  int i=0;
  for(i=0;i<=9;i++)
  printf("Hello World \n");
  exit(0);
}
```

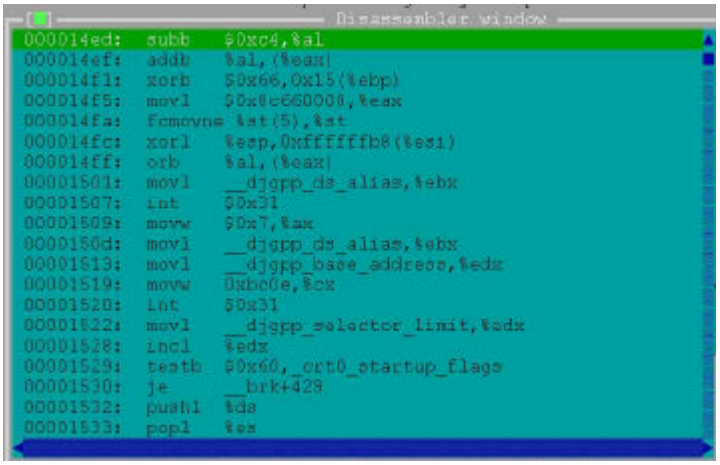


Bild 0.1: Assembler-Sourcecode (das komplette Programm ist ungefähr fünfmal so lang).

Die momentan größte Abstraktionsstufe erreichen die bereits erwähnten visuellen Entwicklungssysteme. Sie erfordern z.T. nicht einmal mehr die Eingabe des Programms als Text, sondern der Programmierer klickt sich seinen gewünschten Programmablauf aus diversen vorgefertigten Elementen auf dem Bildschirm mit der Maus zusammen. Solche Systeme sind meist eng mit der Entwicklung von graphischen Benutzeroberflächen (z.B. Windows®, X11) verknüpft [2].

Spezialisierung bestimmter Programmiersprachen

In diesem Zusammenhang soll auch auf die Spezialisierung einiger Programmiersprachen auf bestimmte Aufgabengebiete hingewiesen werden. Die (ziemlich alte) Programmiersprache FORTRAN z.B., galt und gilt als spezialisiert auf numerische Probleme. Speziell zur Abfrage und Spezifikation von Datenbanken wurden SQL (Structured Queuery Language) und seine zahlreichen Dialekte entwickelt. Letztlich sind die meisten verfügbaren Hochsprachen für die eine oder andere Aufgabe mehr oder weniger geeignet, wobei die Wahl der „richtigen“ Programmiersprache oft durch persönliche Vorlieben, bzw. Vorgaben aus dem EDV-technischen Umfeld bestimmt wird.

Der Turnaround Zyklus bei einer Compilersprache

Die in diesem Skript behandelte Programmiersprache C stellt einen leistungsfähigen und sehr weit verbreiteten Kompromiß zwischen einer Hochsprache und Assembler dar. Sie wurde ursprünglich dazu entworfen das Betriebsystem UNIX , das bis zu diesem Zeitpunkt komplett in Assembler geschrieben war, in einer Hochsprache neu zu programmieren, um unabhängig von der verwendeten Hardware zu werden. Bei der Implementierung eines Betriebssystems kommt es sowohl auf äußerste Effizienz an, als auch auf die elegante und übersichtliche Handhabung komplexer Datenstrukturen. Beide Anforderungen werden von C in ausreichendem Maße erfüllt, wenn man über die zweifellos nötige Erfahrung verfügt.
Da eine Hochsprache, wie bereits erwähnt, nicht direkt durch den Rechner verstanden werden kann, ist eine „Übersetzung“ in Maschinensprache notwendig. Die Art und Weise, wie und wann diese Übersetzung vonstatten geht, eröffnet eine weitere Differenzierungsmöglichkeit für Programmiersprachen. Man unterscheidet nämlich sogenannte „Compilersprachen“ von „Interpretersprachen“. Bei Compilersprachen ist der Ablauf während der Programmierung wie folgt:
Der Programmierer schreibt seinen Programmtext, d.h. die Abfolge der Befehle in der jeweiligen Programmiersprache mit einem sogenannten „Editor“. Dieser Editor kann ein Textverarbeitungsprogramm, wie z.B. Microsoft Word™, oder aber eine speziell für diesen Zweck entwickelte Software (z.B. EMACS bei UNIX Systemen) sein. Das Ergebnis dieses ersten Schrittes ist eine Textdatei, die den sogenannten „Sourcecode“, oder deutsch „Quellcode“ enthält.
Im nächsten Schritt wird die vom Programmierer erstellte Textdatei mit dem Sourcecode von einem weiteren Programm, dem sogenannten Compiler, Zeile für Zeile eingelesen und in Maschinensprache übersetzt. Während des als Compilierung bezeichneten Prozesses gibt der Compiler Fehlermeldungen

aus, falls der Programmierer syntaktische (z.B. Semikolon fehlt) oder semantische Fehler (z.B. das Programm gibt 11 Zeilen Text aus, soll aber nur zehn Zeilen ausgeben) erzeugt hat. Ist ein solcher Fehler aufgetreten, so bricht der Übersetzungsvorgang mit einer entsprechenden Fehlermeldung ab und der Programmierer begibt sich wieder zu Schritt 1 um den Fehler zu korrigieren. Die Compilierung liefert als Ausgabe eine Objektdatei (Objectfile), welche im Gegensatz zum Sourcefile keinen lesbaren Text enthält, sondern binär dargestellte Maschinenanweisungen. In aller Regel übersetzt ein Compiler den Sourcecode in mehreren Stufen („Passes“), die dem Programmierer allerdings in heutigen Systemen weitgehend verborgen bleiben.
Es folgt bei den meisten Entwicklungsumgebungen (s.u.) ein dritter Schritt: der „Linker“ (deutsch: „Binder“). Der Linker ist ein Werkzeug, das mehrere Objectfiles zu einem gesamten ausführbaren Programm zusammensetzt. Es ist häufig so, daß eine Programmieraufgabe aus Gründen der Übersichtlichkeit und Modularisierung in mehrere kleinere und damit leichter handhabbare Teile zerlegt wird, welche dann einzeln entwickelt und anschließend „zusammengelinkt“ werden. Als Eingabe für diesen Schritt dienen die durch Schritt 2 erzeugten Objectfiles und als Ausgabe erhält man schließlich das fertige, binäre Programm, das dann ausgeführt und getestet werden kann. Auch in diesem Schritt werden u.U. Fehler ausgegeben, die oft mit fehlenden Teilen des Gesamtprogramms zusammenhängen und ebenfalls zu einem Rücksprung in Schritt 1 führen.
Die vierte Phase ist das sogenannte Debugging, bzw. das Testen des Programms. Es kommt in der Praxis so gut wie nie vor, daß ein neu entwickeltes Programm sich so verhält, wie es geplant war. Es kommt zu Abstürzen, falschen Ausgaben usw. Um diese Fehler zu entdecken, ist es unabdingbar ein Werkzeug zur Verfügung zu haben, das die Fehlersuche erleichtert. Ein solches Werkzeug ist der Debugger, der ein fertig übersetztes Programm als Eingabe erhält. Nach Start des zu untersuchenden Programms „im Debugger“ kann man (je nach Komfort) Zeile für Zeile den Quellcode abarbeiten lassen, Speicherbereiche anschauen und das Programm genau inspizieren. Sollte der Fehler gefunden werden, beginnt die Programmierung wieder bei Schritt 1. Üblicherweise hält man sich bei der Entwicklung, vor allem von komplexeren Systemen, die mit Abstand meiste Zeit in der Debugging-Phase auf, da die Fehlersuche den größten Teil der gesamten Programmierzeit benötigt.
Nach Durchlauf aller Schritte ist man schließlich beim lauffähigen Programm angelangt, daß man nun auch ohne den Debugger („stand alone“) ausführen kann. Es soll nicht verschwiegen werden, daß der eigentlichen Programmierphase, bei komplexeren Softwaresystemen, eine ausgiebige „Analyse- und Designphase“ vorausgeht. Hier wird die Struktur des gesamten Programms festgelegt und die Interaktion einzelner Programmteile untereinander beschrieben. Diese Vorphasen bei der Software-Entwicklung sind meist wesentlich umfangreicher als die eigentliche Programmierung; sie sind aber nicht Bestandteil dieses Skriptes (s.a. [3]).
Der gesamte Ablauf der Programmierung mit einer Compilersprache ist noch einmal in folgendem Bild zusammengefaßt:

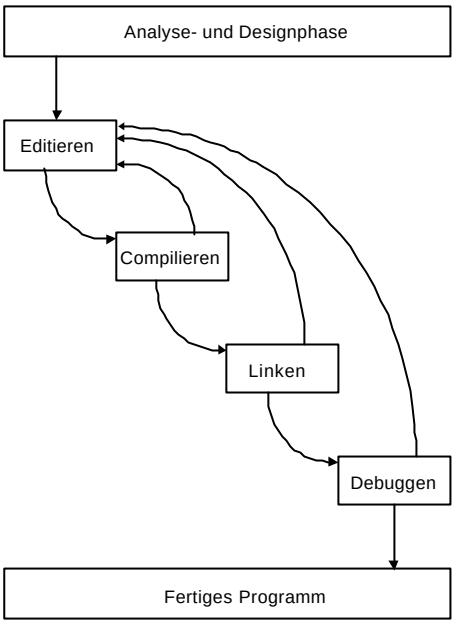


Bild 0.2: Editier-Compilier-Link-Debug-Zyklus.

Der gerade skizzierte Ablauf ist typisch für den Entwicklungsprozeß bei Benutzung einer Compilersprache. Er wird „Turnaround“ genannt und die Zeit, die für einen Durchlauf benötigt wird, ist als „Turnaround-Zeit“ bekannt. Je kürzer diese Zeit ist, desto effizienter sind die Entwicklungswerkzeuge (Compiler, Linker etc.). Im Prinzip stellen die Programme Compiler, Linker, Editor und Debugger eigenständige Programme dar, die einzeln aufgerufen und mit Eingaben versorgt werden müssen. Um den Komfort für den Entwickler zu erhöhen, sind sie heute meistens in einer „Integrierten Entwicklungsumgebung“ oder IDE („Integrated Development Environment“) zusammengefaßt. Eine IDE erleichtert den Umgang mit den einzelnen Komponenten, indem z.B. bei Auftreten eines Fehlers bei der Compilierung direkt der Editor gestartet, das Sourcefile geladen, und die entsprechende Zeile im Sourcefile angezeigt wird. Außerdem bieten IDEs in der Regel weitgehende Möglichkeiten, um große Projekte zu verwalten und graphische Benutzeroberflächen zu implementieren [2].
Die Sprache C ist eine Compilersprache (es gibt auch interpretierte Versionen, die allerdings keine große Verbreitung gefunden haben), im Gegensatz dazu ist z.B. die in der künstlichen Intelligenz verbreitete Sprache LISP eine interpretierte Sprache. Bei einem interpretierten Programm findet der oben beschriebene 2. Schritt synchron zur Programmausführung statt. D.h. jede Anweisung wird erst dann in Maschinensprache übersetzt, wenn das Programm bereits läuft. Tritt ein Fehler auf, so hält die Programmausführung an und der Quellcode kann direkt verbessert werden. Im allgemeinen wird ein solches Vorgehen als benutzerfreundlicher empfunden, wobei es allerdings einige Einschränkungen gibt. Der durch Interpretation erzeugte Maschinencode ist meist ineffizienter als compilierter Code. Außerdem sind leistungsfähige Sprachkonstrukte, wie etwa objektorientierte Datenstrukturen (s. a. C++-Teil dieses Skriptes) entweder gar nicht, oder nur sehr aufwendig zu übersetzen. Durch den Einsatz von IDEs schwinden zunehmend die für den Programmierer sichtbaren Unterschiede zwischen Compiler und Interpreter. Insbesondere bei IDEs zur visuellen Programmierung (z.B. Borland Delphi™) werden auch hybride Verfahren eingesetzt [4].

Übersicht über einige Programmiersprachen

Im folgenden soll eine kurze Übersicht (ohne jeden Anspruch auf Vollständigkeit) über einige gängige Programmiersprachen gegeben werden, wobei die oben vorgestellten Unterschiede zur Geltung kommen.

Tabelle 1: Einige Programmiersprachen und ihre Eigenschaften

Programmier-sprache	Compiler (C)/ Interpreter (I)	Hauptanwendungsbereich	Besonderheiten
BASIC	I und C	Einsteigersprache, inzwischen auch oft zur GUI-Programmierung verwendet	Einfach zu lernen, Inzwischen aber trotzdem sehr mächtig
C/C++	C (sehr selten I)	Vielseitig verwendbar, auch maschinennahe Systemprogrammierung, Grafikprogrammierung	Erlaubt sehr kompakte und effiziente Programmierung verbunden mit hohen Abstraktions-möglichkeiten
COBOL	C	Kaufmännische Programme	Sehr schwerfällig, Sourcecode fast wie Umgangssprache
FORTH	I	Maschinennahe Programmierung	Kryptische Befehle, extrem flexibel und erweiterbar
FORTRAN	C, I	Numerische Problemstellungen	Eingeschränkter Sprachumfang, älteste Hochsprache
LISP	I	Künstliche Intelligenz	Völlig andere Konzepte als bei herkömmlichen Sprachen
PASCAL	C	Lehrsprache, Inzwischen auch Objektorientiert	Strenge Syntax/Semantik
SMALLTALK	C	Objektorientierte Programmierung großer Systeme	Strenge Objektorientierung
SQL	I und C	Spezialsprache zur Datenbankabfrage	Weite Verbreitung, viele Dialekte

Die verwendete IDE

Im Rahmen dieser Veranstaltung wird eine Entwicklungsumgebung, d.h. ein integriertes Paket aus Editor, Compiler, Linker und Debugger der Firma Borland verwendet. Diese Borland C++ 3.0 genannte IDE ist kompatibel mit dem als „Public-Domain“ (freie Software) verfügbarem System DJGPP. Von letzterem stammen die „Bildschirmkopien“, die im folgenden die einzelnen Bedienschritte illustrieren sollen. Die grundsätzliche Bedienung der IDE wird anhand des oben skizzierten Beispiels, d.h. der Ausgabe von zehn Zeilen mit dem Text „Hello World“ dargestellt. Es ist zunächst noch nicht notwendig den Programmtext im einzelnen zu verstehen, er sollte lediglich so übernommen werden, um Fehlermeldungen beim Übersetzen zu verhindern. Beide IDEs sind zusätzlich mit einer leistungsstarken, umfassenden „Online-Hilfe“ ausgestattet, so daß bei weiteren Fragen oder Problemen zunächst diese konsultiert werden kann.

Starten der IDE und Anlegen eines Projektes

Zunächst muß die IDE durch Eingabe von „bc“ (für Borland C++) oder „rhide“ (für DJGPP) auf der Kommandozeile gestartet werden (bei Verwendung von Windows 95® entsprechende Links erzeugen, anklicken und in DOS-Box starten) (Bild 0.3)
Nach dem Start der IDE ist es notwendig ein Projekt anzulegen, indem man den Befehl „Open Project“ aus der Menüleiste anwählt (Bild 0.4). Im sich nun öffnenden Projektfenster kann man eine, oder mehrere Sourcedateien in das Projekt einfügen (Bild 0.5). Das Anlegen eines Projektes ist eigentlich für größere Programmieraufgaben, die aus einer größeren Anzahl an Dateien bestehen können, gedacht. Trotzdem sollte man auch bei kleineren Aufgaben mit dem Anlegen eines Projektes beginnen, um zum einen die Arbeitsweise zu erlernen, und zum anderen alle Möglichkeiten der IDE ausnutzen zu können.

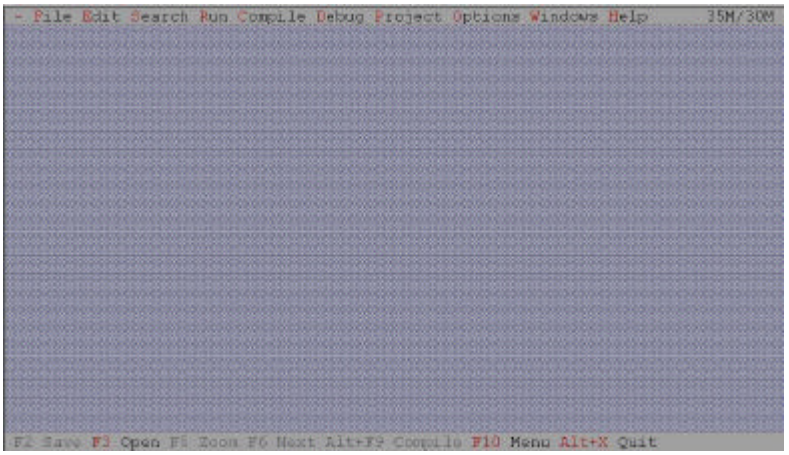

File Edit Search Run Compile Debug Project Options Windows Help
35M/30M
F2 Save F3 Open F5 Zoom F6 Next Alt+F9 Compile F10 Menu Alt+X Quit

Bild 0.3:IDE (leer).

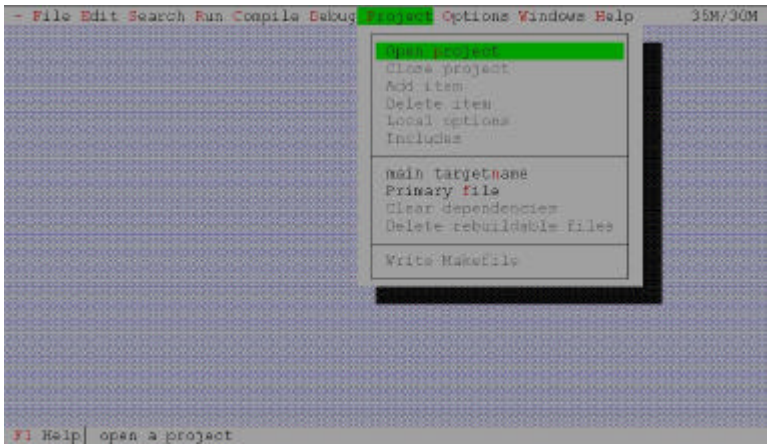


Bild 0.4: „Projekt öffnen“.

File Edit Search Run Compile Debug Project Options Windows Help
35M/30M
Project Window
hello10.c
Ins Add Del Delete ^O Local ^I Includes F5 Zoom Alt+F9 Compile Alt+X Quit

Bild 0.5: Projektfenster.

Eingabe des Quelltextes

Im geöffneten Quelltextfenster kann nun der oben angegebene Quelltext eingegeben werden (Bild 0.6). Dieses Fenster öffnet sich entweder von selbst, beim Hinzufügen einer Sourcedatei zum Projekt, oder es kann durch Doppelklick auf eine Sourcedatei im Projektfenster geöffnet werden). Es ist darauf zu achten, wirklich alle Zeichen genau so einzugeben, wie sie angegeben sind. Insbesondere Leerzeichen sind für eine reibungslose Übersetzung wichtig. Man beachte auch hier die „Online-Hilfe“, der IDE um die genaue Tastenbelegung des Editors kennenzulernen. Ist die Eingabe beendet, sollte man das Abspeichern mit F2 nicht vergessen, da es gerade bei der Programmierung zu Abstürzen kommen kann, die die gerade eingegebenen Daten vernichten können.

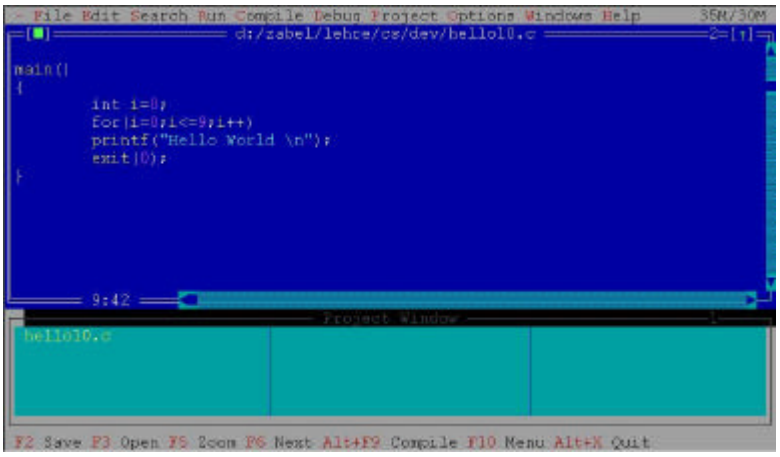


Bild 0.6: Texteditor.

Übersetzen und Linken

Durch Anwahl des Menüpunktes „Make“, oder drücken der Taste F9 wird nun der Übersetzungsprozeß angestoßen (Bild 0.7). Der integrierte Compiler übersetzt den Sourcecode in den Objectcode und meldet evtl. aufgetretene Fehler im Ausgabe-Fenster (Bild 0.8). Sollte ein Fehler aufgetreten sein, kann man durch Doppelklick auf die entsprechende Zeile einfach in das entsprechende Sourcecode-Fenster genau an die Stelle, an der der Fehler aufgetreten ist, gelangen. Hier kann man die nötigen Korrekturen durchführen.

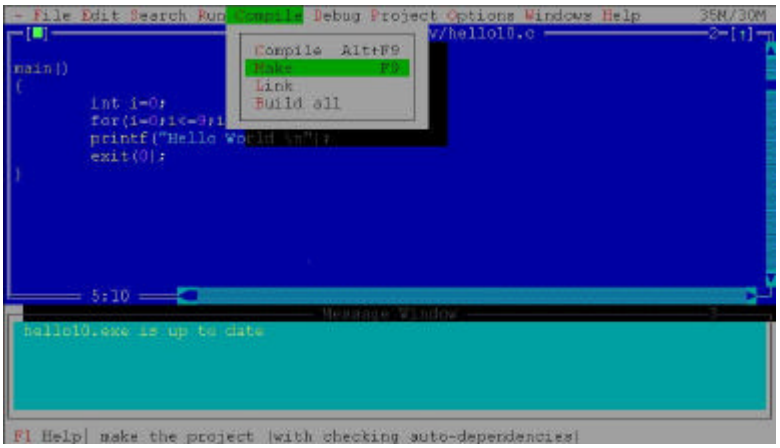


Bild 0.7 Make.

Der „Link“-Vorgang ist durch Auswahl des „Make“-Befehls aus der Menüleiste nicht mehr als eigenständiger Schritt zu erkennen, er wird bei fehlerfreier Compilierung automatisch ausgeführt und erzeugt das lauffähige Programm. Wählt man statt des „Make“-Befehls den „Compile“-Befehl (ALT-F9), so wird lediglich der Sourcecode des aktuellen Fensters übersetzt, aber kein Linking durchgeführt und es werden auch keine evtl. vorhandenen anderen Sourcefiles compiliert. An dieser Stelle machen sich bereits die Vorteile einer IDE bemerkbar: Es werden nämlich nur solche Sourcefiles neu übersetzt, die vom Programmierer auch verändert worden sind und nicht etwa alle Sourcefiles in einem Projekt. Ein solches Vorgehen muß bei autonomen Kommandozeilenprogrammen mühsam vom Programmierer initiiert und kontrolliert werden. Zur Verdeutlichung des Übersetzungsvorgangs sind die wesentlichen Schritte in den folgenden Screenshots festgehalten.

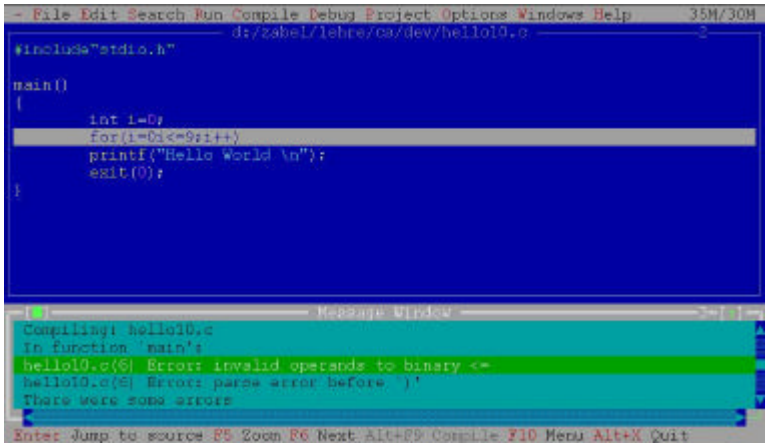


Bild 0.8 Debugging.

Ausführen und Debugging des Programms

An dieser Stelle ist ein ausführbares Programm von der IDE erstellt worden. Dieses Programm kann man nun starten, indem man den „Run“-Befehl aus der Menüleiste anwählt. Beim Start des Programms wechselt der Bildschirm auf den Ausgabebildschirm und man sieht das Ergebnis des Programms, nämlich zehn Zeilen mit dem Text „Hello World“. (Bild 0.9, Bild 0.10). Da am Ende des Programms kein expliziter Stop-Befehl angegeben wurde, erscheint der Ausgabebildschirm nur ganz kurz. Man kann sich diesen Bildschirm durch Wahl des Befehls „User Screen“ aus dem Menüeintrag „Windows“ aber erneut ansehen.

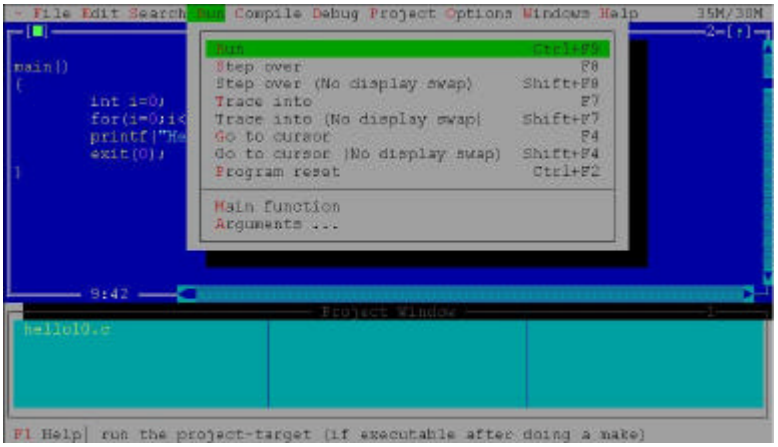

File Edit Search Run Compile Debug Project Options Windows Help
Run Ctrl+F9
Step over F8
Step over (No display swap) Shift+F8
Trace into F7
Trace into (No display swap) Shift+F7
Go to cursor F4
Go to cursor (No display swap) Shift+F4
Program reset Ctrl+F2
Main function
Arguments ...
Project Window
hello10.c
F1 Help] run the project-target (if executable after doing a make)

Bild 0.9: Hello World 1.

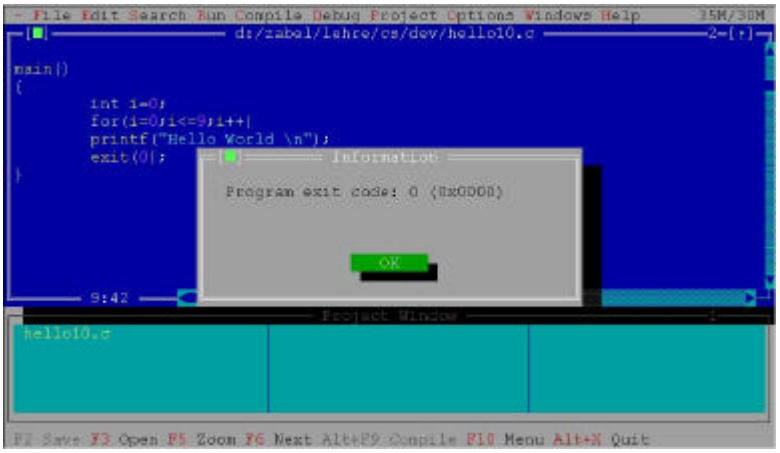


Bild 0.10: Hello World 2.

Zum Debugging sind einige weitere Schritte notwendig. Das Programm wird im Prinzip genauso gestartet, wie bei der einfachen Ausführung, allerdings gibt man einen Punkt, oder besser gesagt eine Zeile im Sourcecode vor, an dem die Programmausführung zunächst anhalten soll. Diese Zeile ist vor dem Starten im Sourcecode-Fenster durch „Set / Reset-Breakpoint“ aus dem Menü „Debug“ oder drücken von CTRL-F8 zu markieren (Bild 0.11). Startet man das Programm jetzt wie oben beschrieben, so wird die Ausführung genau an dieser Stelle angehalten. Der Bildschirm wechselt nur kurz auf den Ausgabebildschirm, um dann wieder zurück in das Sourcecodefenster zu wechseln. Jetzt hat man die Möglichkeit sich einzelne Daten über das gerade laufende Programm anzusehen, was an dieser Stelle nur exemplarisch für die Laufvariable „i“ durchgeführt werden soll (s. Bild 0.12, Bild 0.13). Außerdem kann man die Programmausführung Schritt für Schritt durchführen, indem mit der Taste F8 Zeile für Zeile weitergeht. Während des schrittweisen Weitergehens ist es möglich, bestimmte Werte in einem separaten Fenster zu betrachten, was an dieser Stelle ebenfalls wieder für die Variable „i“ unter Benutzung des „Watch“-Befehls aus dem Menü: „Debug“ erfolgen soll.

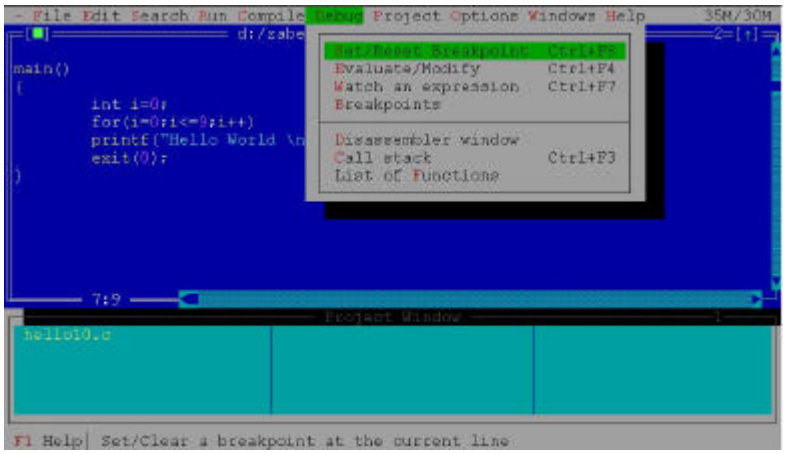


Bild 0.11: Variablen „watching“ (1)

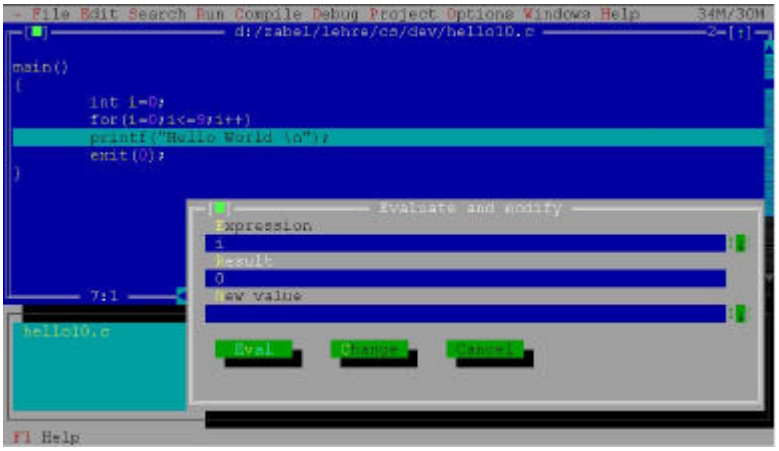


Bild 0.12: Variablen „watching“ (2)

Bild 0.13: Ausführungszeile.

Damit hat man die grundlegenden Schritte vom Editieren bis zum Testen eines neu zu entwickelnden Programmes zunächst einmal durchgeführt. Die Vorgehensweise unterscheidet sich auch bei anderen C/C++ - IDEs (z.B. Microsoft Visual C++ [2]) im Prinzip nicht von dem hier skizzierten. Natürlich kann und soll ein solches Skript wie das hier vorliegende keine erschöpfende Bedienungsanleitung darstellen, sondern vielmehr Konzepte vermitteln, die als Grundlage zur C-Programmierung verstanden werden müssen.
Für weitergehende Informationen sei zum einen auf die Online-Hilfen der im Skript vorgestellten IDEs verwiesen und zum anderen auf weiterführende Literatur, wie z.B. [5] und [6].
Zur Programmiersprache C sind die Werke [7] und [8] empfehlenswert, wobei es natürlich noch eine große Menge anderer lesenswerter Literatur, gerade zu diesem Thema, gibt. Grundsätzliche Dinge zum Rechneraufbau und zur Rechnerbenutzung sind in [9] zu finden.

Grundbegriffe und Sprachstrukturen

Bezeichner

Bezeichner sind Namen für beliebige "Objekte" im Programm: Variablen, Funktionen, selbstdefinierte Datentypen, Klassen, Objekte etc. Bezeichner sind aus beliebigen Buchstaben (keine deutschen Umlaute!), Ziffern und dem Zeichen _ (Underscore) zusammengesetzt, wobei allerdings das erste Zeichen keine Ziffer sein darf.
Beispiel für gültige Bezeichner sind hallo, _hallo, hallo123. Ungültig sind 123hallo,+hallo oder ätsch.
Weiterhin sind zwei Gesichtspunkte zu beachten:

- C und C++ sind Case Sensitive, d.h. es unterscheidet zwischen Groß- und Kleinschreibung. Hello, Hello und HeLlO sind also unterschiedliche Bezeichner.
- Wörter werden durch Zeichen, die keine Buchstaben, Ziffern oder Underscores sind, begrenzt. Daher ist zum Beispiel elseif ein einzelner Bezeichner und nicht die Aufeinanderfolge von `else` und `if`.

Bei der Wahl der Bezeichnernamen ist es wichtig, konsistent vorzugehen. Sie sollten nach bestimmten (frei wählbaren) Regeln aufgebaut werden damit bereits aus dem Namen zu erkennen ist, um was für eine Art von Bezeichnern es sich handelt. Die verschiedenen Compilerhersteller und Autoren von Büchern zum Thema C und C++ haben eigene "Richtlinien" für die Schreibweise herausgegeben. Einige dieser Regeln haben sich als nützlich herausgestellt und zu einem Quasi-Standard herausgebildet:

- Namen von Variablen und Funktionen (bei C++ auch Methoden) beginnen mit einem Kleinbuchstaben, bei zusammengesetzten Wörtern beginnt ein neuer Wortteil mit einem Großbuchstaben, wie bei eineGrosseZahl.
- Konstantennamen werden groß geschrieben, wobei bei zusammengesetzten Wörtern die Wortbestandteile mit einem Underscore verbunden werden, wie beispielsweise bei MAXIMALE_LAENGE.
- Selbstdefinierte Typennamen (Namen von Strukturen, Aufzählungstypen oder Klassen in C++) beginnen mit einen Großbuchstaben und werden ansonsten wie Variablen benannt.

Schlüsselwörter

C und C++ zeichnen sich - verglichen mit anderen Programmiersprachen - durch eine sehr begrenzte Anzahl reservierter Bezeichner, sog. Schlüsselwörter, aus. Diese haben bereits eine vordefinierte Bedeutung und dienen zur Beschreibung von Aktionen, vordefinierten Datentypen und Objekten (C++). Sie dürfen daher nicht anderweitig verwendet werden. Die folgende Tabelle gibt eine Übersicht über die Schlüsselwörter von C und C++.

Tabelle 2: Schlüsselwörter in C und C++ (kursiv)

Kategorie	

Basisdatentypen	*bool true false* char *wchar_t* short signed unsigned int long float double void
Datentypen	enum *class* struct union typedef
Speicherklassen	auto extern register static *mutable*
Qualifizierer	const volatile
Kontrollstrukturen	if while else case switch default do for
Sprunganweisungen	break continue return goto
Typumwandlungen	*const_cast dynamic_cast reinterpret_cast static_cast*
Typsystem	*typeid typename*
Namensräume	*namespace using*
Operatoren	*new delete* sizeof *operator this*
Zugriffsschutz	*friend private public protected*
Funktionsattribute	*inline virtual explicit*
Generizität	*template*
Ausnahmebehandlung	*catch throw try*
Assemblercode	asm

Neben diesen Standard-Schlüsselwörtern existieren noch andere reservierte Wörter, die alternativ zu den Symbolen für Operatoren (Operator-Symbole) verwendet werden können.

Aufbau eines C-Programms

Ein C Programm gliedert sich in aller Regel in folgende Bereiche auf:

Präprozessor Anweisungen zum Einfügen von Header-Dateien

Deklaration von Funktionen und globalen Variablen

Definition, d.h. die Implementierung, der zuvor deklarierten Funktionen.

Von den genannten Bereichen ist nur einer in jedem C-Programm zwingend erforderlich: Die Implementation einer Funktion, und zwar der Funktion `main`. Wie der Name vermuten läßt handelt es sich hierbei um eine Art "Hauptprogramm". Die Notwendigkeit der anderen Bereiche hängt von der jeweiligen Programmieraufgabe und insbesondere den in der main-Funktion verwandten Variablen, Typen und Funktionen.

Einbinden von Header-Dateien

Bei umfangreichen Programmieraufgaben empfielt es sich, das Programm zu modularisieren. Dazu werden die verschiedenen Funktionen auf mehrere Quellcodedateien aufgeteilt. Um die in einer Datei definierten Funktionen aber in einer anderen Datei nutzen zu können, müssen in der "nutzenden" Datei die Funktionsnamen bzw. die Deklaration der Funktionen bekannt gemacht werden. Hierzu definiert man zu einer Quellcodedatei (Dateinamenserweiterung .c) eine sog. Header-Datei (Dateinamenserweiterung .h). In dieser Header-Datei werden alle Funktionen und Variablen, die auch von außerhalb der Quellcodedatei aufrufbar sein sollen, deklariert. Wie eine solche Deklaration erfolgt, wird in Kapitel 0 erläutert. Die Header-Datei muß nun mit Hilfe einer sog. Präprozessor-Anweisung, dem Befehl `#include <dateiname.h>` in die aufrufende Datei eingebunden werden. Der Präprozessor führt - wie der Name vermuten läßt - vor dem eigentlichen Compilieren des Programmcodes einen "Vorlauf" durch, bei dem u.a. die `#include`-Anweisungen durch die jeweils angegebenen Header-Dateien ersetzt wird (reine Textersetzung!). Durch dieses Einbinden der Header-Datei sind alle in ihr deklarierten Funktionen bekannt und können benutzt werden.

Die Funktion main

Wie bereits angedeutet ist die Funktion `main` in jedem C-Programm erforderlich. Jedes Programm beginnt seine Ausführung am Anfang von `main`. Die Funktion `main` wird normalerweise andere Funktionen aufrufen, um ihre Aufgaben zu erledigen: solche Funktionen können aus dem gleichen

Programm kommen oder aus vorher entwickelten oder vom Compiler-Hersteller mitgelieferten Funktionsbibliotheken. Da C an sich nur einen sehr begrenzten Sprachumfang hat (siehe Schlüsselwörter), ist ein reger Zugriff auf solche Bibliotheken notwendig.
Die `main`-Funktion besteht wie jede Funktion aus dem Funktionskopf, in dem die Übergabeparameter und der Rückgabewert definiert werden, und dem Funktionsblock, der von geschweiften Klammern begrenzt wird. Ein Block ist eine "Ansammlung" von Anweisungen, die durch die Klammerung quasi zu einer Anweisung zusammengefaßt werden.
Das unten stehende Beispiel zeigt den prinzipiellen Aufbau eines einfachen C-Programms. Wie beim ersten Programm einer neu zu erlernenden Programmiersprache üblich, gibt dieses Programm den Text "hello world" auf dem Bildschirm aus. Hierzu wird die Funktion `printf` benötigt, die in einer Standard-Funktionsbibliothek definiert und in der Header-Datei `stdio.h` deklariert ist. Daher muß diese Header-Datei mittels `#include`-Anweisung eingebunden werden, um die Funktion `printf` im Programm bekannt zu machen und verwenden zu können. Diese Funktion gibt den in Klammern angegebenen Text auf dem Bildschirm aus. Die Zeichenkombination `\n` ist ein sogenanntes Steuerzeichen, das dafür sorgt, daß der nächste auszugebende Text in der nächsten Zeile am Zeilenanfang erscheint (wie wenn man in einem Texteditor die Eingabe-Taste betätigt):

```
#include <stdio.h>


main()
{
  printf ("hello, world \n");
}
```

Variablen - Einfache Datentypen

Bei der praktischen Umsetzung von Problemstellungen in ein Computerprogramm spielen benutzerdefinierte Variablen und Konstanten, über die der Anwender frei verfügen kann, eine zentrale Rolle. Solchen Variablen können während der Programmausführung (der sog. „Laufzeit“ oder engl. „runtime“) Inhalte zugewiesen und zu einem späteren Zeitpunkt wieder ausgelesen werden. Hierzu werden die Inhalte zwischenzeitlich „irgendwo im Speicher des Rechners“ abgelegt. Wo dies geschieht, hängt von der Art und Weise ab, wie die Variablen erzeugt werden. Die vollständige (stilistisch saubere) Erzeugung einer Variable unterteilt sich in zwei Abschnitte:

- Definition
- Initialisierung

Unter Definition versteht man die Bekanntmachung des „Namens“, den man für die Variable vorgesehen hat und unter der Initialisierung die Zuweisung eines „ersten Inhaltes“ in den zugehörigen Speicherbereich.
Um eine Variable zu erzeugen, müssen zunächst drei wichtige Informationen bekannt sein.

- Wie soll die Variable heißen? (Variablennamen)
- Was für Information soll in der Variablen gespeichert werden? (Variablentyp)
- Welchen Inhalt soll die Variable nach ihrer Erzeugung erhalten? (Initialisierungswert)

Variablennamen

Bei der Vergabe von Variablennamen sind einige syntaktische und stilistische Aspekte zu berücksichtigen. Die syntaktischen Vorschriften sind verbindlich und zum Teil auch von dem verwendeten Compiler abhängig, während die stilistischen Aspekte, die weiter unten erläutert werden, im Laufe der Zeit gewachsen sind und die Lesbarkeit des C++ Quellcodes verbessern. Syntaktisch gelten die folgenden Grundregeln [7]:

- Namen bestehen aus Buchstaben und Ziffern
- Das erste Zeichen muß ein Buchstabe sein
- Das Unterstrichzeichen _ gilt als Buchstabe
- Nur die ersten acht Zeichen eines Namens sind signifikant (es können aber durchaus mehr Zeichen benutzt werden).
- Es dürfen keine Schlüsselwörter als Namen vergeben werden
- Groß- und Kleinbuchstaben werden unterschieden.

Diese allgemeinen Regeln müssen immer berücksichtigt werden, während die folgenden Richtlinien im wesentlichen die Lesbarkeit des Codes fördern:

- Variablennamen sollten eine Bedeutung erkennen lassen (hierbei erhöht insbesondere die Verwendung des Zeichens _ die Lesbarkeit von langen Namen).

- Variablennamen werden im allgemeinen klein begonnen und gemäß der gängigen Schreibweise fortgesetzt, während Konstanten (bis auf die Typkennzeichnung) groß geschrieben werden.
- Dem eigentlichen Variablennamen wird ein Kürzel für den Variablentyp vorangestellt.

Zu diesen Regeln einige Beispiele:

Negativbeispiele:

`?§Name` ungültig, weil kein Buchstabe am Anfang.
`1_Name` ungültig, weil eine Zahl am Anfang steht.
`VARIABLEeins`, `VARIABLEzwei` sind nicht unterscheidbar, weil sie in den ersten acht Buchstaben identisch sind.

Musterbeispiele:

`lpctstr_Vorname_1` Die ersten sieben Buchstaben charakterisieren den Variablentyp (lpctstr := **l**ong **p**ointer to a **c**onstan**t str**ing), danach wird der Inhalt „Vorname 1“ beschieben.
`n_Alter` n zeigt an, daß es sich bei der Variable um einen Integerwert handelt (n, wie **N**ummer, vgl. Variablentyp), der das Alter beinhaltet.
`f_PI` f symbolisiert, daß es sich hierbei um einen Fließkommawert handelt, die Großschreibung, daß es eine Konstante ist und PI steht für die Kreiskonstante π.
Durch die Ergänzung des Variablennamens mit einer vorangestellten Typkennzeichnung kann während der Programmierung leicht erkannt werden, ob evtl. ungültige Zuweisungen durchgeführt werden, wie zum Beispiel:

```
n_Alter = 14,3;
```

Es ist sofort ersichtlich, daß die Zuweisung des Fließkommawertes „14,3„ dem Typ der Variable `n_Alter` (Integer!) wiederspricht.

Datentypen

Unter dem Datentyp versteht man die Art des Inhaltes einer Variablen. In der Programmiersprache C gibt es nur sehr wenige elementare Datentypen, aus denen jedoch beliebig viele sog. komplexe Datentypen zusammengesetzt werden können. (s. Kapitel 0). Im folgenden sind die vier implementierten Grundtypen und die dazugehörigen Speicherbereiche aufgelistet:

Tabelle 3: Grundtypen der Programmiersprache C.

`char`	Ein Byte, kann ein Zeichen aus dem Zeichensatz der Maschine aufnehmen
`int`	Ganzzahliger Wert, in der für die Maschine „natürlichen“ Größe
`float`	Einfach genauer Gleitkommawert
`double`	Doppelt genauer Gleitkommawert

Diese Grundtypen können je nach Bedarf noch durch sogenannte „Qualifizierer“ beeinflußt werden. Hierzu stehen die drei Schlüsselwörter `short`, `long` und `unsigned` zur Verfügung.
Der Wertebereich, den die verschiedenen Typen belegen, ist von dem jeweiligen Betriebssystem bzw. vom Computertyp abhängig. Für das 32 Bit-Betriebssystem Windows 95® beispielsweise gelten die in Tabelle 4 Werte.

Tabelle 4: Durch Modifizierer erweiterte einfache Datentypen.

Typ	Anzahl der Bytes	Wertebereich
char	1	-128 bis 127
unsigned char	1	0 bis 255
int	4	-2,147,483,648 bis 2,147,483,647
short int	2	–32,768 bis 32,767
long int	4	-2,147,483,648 bis 2,147,483,647
Unsigned int	4	0 bis 4,294,967,295
float	4	± 3.4E ± 38 (7 digits)
double	8	± 1.7E ± 308 (15 digits)
Long double	10	± 1.2E ± 4932 (19 digits)

Deklaration

Um eine Variable zu Erzeugen, muß sie dem System zuerst bekannt gemacht werden. Dieser Vorgang wird Deklaration genannt. Die Deklaration setzt sich syntaktisch aus einem Variablentyp und dem Variablennamen zusammen, die durch einen Zwischenraum getrennt sind [7].

Beispiel:

```
int n_Nummer;
```

Es können auch mehrere Variablen des gleichen Typs in einer Zeile kommasepariert deklariert werden, wie zum Beispiel:

```
double dVariable_1, dVariable_2, dVariable_3;
```

Äquivalent zu der vorhergehenden Liste können alle Variablen auch einzeln deklariert werden:

```
double dVariable_1;       // Durchmesser des ersten
                          // Teilkreises
double dVariable_2;       // Rohlingsdurchmesser
double dVariable_3;       // Länge des Flansches
```

Der Vorteil liegt hierbei in der Übersichtlichkeit und der Möglichkeit, zu jeder Variablen einen Kommentar hinzuzufügen.
Variablen können auch innerhalb der Deklaration initialisiert werden, indem nach dem Variablennamen ein Gleichheitszeichen und eine Konstante angegeben wird.

Beispiel:

```
double dVariable_1 = 1.4;
char   cChar     = 'A';
char   lpstrString[20] = "Ich bin ein String";
```

Durch die Verwendung der eckigen Klammern „[“ und „]“ kann von jedem Variablentyp eine eindimensionales Feld (Array) erzeugt werden. Die einzelnen Komponenten können wiederum durch eckige Klammern mit einer Komponentenangabe angesprochen werden. **Achtung; die Variablen bezeichnen in diesem Fall einen Zeiger des entsprechenden Typs, nicht den Inhalt!** (vergleiche Kapitel 0)

Beispiel:

```
int main()
{
  int Vektor [12]; // Vektor mit 12 Integerwerten

  Vektor[0] = 14;
  Vektor[1] = -4;
  Vektor[2] = 115;
  .
  .
  .
  return 0;
}
```

Der Bezeichner `Vektor` ist also vom Typ `int*` und nicht vom Typ `int`, während hingegen durch Verwendung der Klammern `Vektor` wieder dereferenziert werden kann und somit `Vektor[2]` wieder vom Typ `int` ist.

Kontrollstrukturen

Unter dem Begriff Kontrollstrukturen versteht man alle Arten von Anweisungen, die einen Einfluß auf die Reihenfolge nehmen, in der ein Quellcode abgearbeitet wird. Hierzu gehören Schleifenstrukturen, Verzweigungen und Bedingungsketten. Der Einflußbereich einer Kontrollstruktur ist in der Regel durch den zur Anweisung gehörigen Block begrenzt. Vor den weiteren Ausführungen sollen deshalb noch einmal kurz die beiden Begriffe "Anweisung" und "Block" ins Gedächtnis zurückgerufen werden. Ein Ausdruck wie zum Beispiel;

```
n = n + 5
```

wird durch ein nachgestelltes Semikolon `;` zu einer Anweisung (das Semikolon beendet die Anweisung):

```
n = n + 5;
```

Mehrere Anweisungen und/oder Vereinbarungen (Deklarationen, Definitionen) können durch geschweifte Klammern zu einem Block zusammengefaßt werden.
Beispiel:

```
{                       /* Anfang des Blocks    */
  int  n = 0;           /* 1. Vereinbarung      */
  int  m = 1;           /* 2. Vereinbarung      */
  n = n + 5;            /* 1. Anweisung         */
  m = m + 2;            /* 2. Anweisung         */
}                       /* Ende des Blocks      */
```

Ein solcher Block ist syntaktisch äquivalent zu einer einzelnen Anweisung. Nach der rechten Klammer steht kein Semikolon. In jedem Block können Variablen vereinbart werden.

"if"-Anweisung

Die `if`-Anweisung erzeugt eine Verzweigungsstruktur die durch einen Entscheidungsausdruck eingeleitet wird. Es gilt die folgende Syntax:

```
if (Ausdruck)
{
  // if-Zweig
}
```

Für den Fall, daß die Auswertung des Ausdrucks „0“ ergibt, wird der `if`-Zweig übersprungen und die Programmausführung unmittelbar hinter diesem fortgesetzt. Möchte man einen Block erzeugen, der nur (ausschließlich) im Fall einer negativen Auswertung des Ausdrucks abgearbeitet wird, so steht hierfür die `else`- Anweisung zur verfügung.

„else“-Anweisung

Die `else`-Anweisung kann nach folgender Syntax an einen „if“-Block angehängt werden:

```
if (Ausdruck)
{
  // if-Zweig
}
else
```

```
{
  // else-Zweig
}
```

Natürlich lassen sich nach diesem Muster beliebig viele „if – else“ Blöcke aneinander hängen. Man spricht dann von sogenannten „else-if“-Ketten.

Beispiel:

```
if(AmpelROT){
  BleibStehen();
}
else{
  if(StrasseWirklichFrei){
       GeheÜberDieStrasse();
  }
  else{
       if(AutofahrerIstBeiRotGefahren){
            MeckernUndStehenbleiben();
       }
       else{
            NichtMeckernUndStehenbleiben();
       }
  }
}
```

„switch“-Anweisung

Die `switch`-Anweisung ist eine besondere Art von Auswahl unter mehreren Alternativen. Hier wird untersucht, ob ein Ausdruck einen von mehreren konstanten Werten besitzt. Ist dies der Fall, so wird entsprechend verzweigt. Die Syntax hierzu lautet:

```
switch(Integer){
  case(Wert1):
       Statements
  case(Wert2):
       Statements
  case(Wert3):
       Statements
  case(Wert4):
```

```
        Statements
    case(Wert5):
        Statements
    case(Wert6):
        Statements
    case(Wert7):
        Statements
}
```

Die Ausdrücke Wert1...Wert7 beziehen sich hierbei auf den Inhalt des Integers. Die `case`-Marken sind hierbei lediglich als Textmarken zu interpretieren. Das bedeutet, daß bei der obigen Konstellation im Falle einer Verzweigung nach „Wert4“, auch die Statements unter Wert5 bis Wert7 abgearbeitet werden. Um dies zu verhindern und nach Abarbeiten eines Statements den gesamten `switch`-Block zu verlassen, muß am Ende der Statements die `break`-Anweisung stehen.

„break“-Anweisung

Die `break`-Anweisung führt dazu, daß der aktuelle Block (geschweifte Klammern!) unmittelbar verlassen wird. Die Syntax lautet:

```
break;
```

Unter Verwendung dieses Schlüsselwortes kann jetzt der obige `switch`-Block so geschrieben werden, wie er eigentlich gedacht war:

```
switch(Integer){
    case(Wert1):
        Statements
        break;
    case(Wert2):
        Statements
        break;
    case(Wert3):
        Statements
        break;
    case(Wert4):
        Statements
        break;
    case(Wert5):
        Statements
        break;
    case(Wert6):
```

```
        Statements
        break;
    case(Wert7):
        Statements;
        break;
    default:
        default_Statements;
        break;
}
```

Letzlich nehmen noch die Schleifen eine wichtige Position unter den Strukturelementen ein.

„while“und „for“-Schleife

Die Syntax diese beiden Schleifen lautet:

```
for(Ausdruck1; Ausdruck2; Ausdruck3)
{
  Statements;
}
```

beziehungsweise für die `while`-Schleife:

```
while(Ausdruck2)
{
  Statements;
}
```

`Ausdruck1` und `Ausdruck3` sind entweder Zuweisungen oder Funktionsaufrufe während hingegen `Ausdruck2` eine Bedingung ist.

Beispiele:

```
for (int i = 0; i < 20; i++)
{
  printf(„Die Zahl \“i\“ ist jetzt: %d“, i);
}

int i = 0;
while(i < 20)
{
  printf(„Die Zahl \“i\“ ist jetzt: %d“, i);
}
```

Beide Konstrollstrukturen haben die Eigenschaft, daß die Abbruchbedingung vor der Schleife steht. Möchte man nun aber mindestens einmal die Schleife durchlaufen (selbst wenn die Abbruchbedingung bereits erfüllt ist) so kann man das `do`-Statement benutzen.

„do“-Anweisung

`do` arbeitet immer im Zusammenhang mit `while` und sorgt dafür, daß die Abbruchbedingung erst nach dem Durchlaufen der Schleife geprüft wird. Die Syntax lautet dann:

```
int i = 0;
do
{
  printf(„Die Zahl \“i\“ ist jetzt: %d“, i);
} while(i < 20)
```

Zeiger

Um eine Variable verwalten zu können, muß das System ihren Wert, ihre Größe (den Speicherbedarf) und den Platz (die Speicheradresse), an der die Variable im Speicher abgelegt ist, kennen. Das Problem dabei ist, daß durch die Deklaration Umfang und Struktur der Daten fix vorgegeben sind. Damit müssen bereits zum Zeitpunkt der Programmerstellung alle Angaben bezüglich Größe und Art der zu verwaltenden Daten festgelegt werden. In der Praxis ist aber im Vorhinein oft nicht bekannt, wie groß die zu bearbeitende Datenmenge ist, bzw. es ist Flexibilität hinsichtlich der Datenhaltung gewünscht.
Eine Möglichkeit, diese Einschränkung zu umgehen, ist es, Variablen über ihre Adresse im Speicher zu verwalten. Dazu werden sogenannte Zeiger verwendet.
Unter einem "Zeiger" versteht man eine Variable, die als Wert die Speicheradresse einer anderen Variable enthält, man sagt: Sie zeigt auf eine andere Variable. Zeiger werden in C sehr häufig benutzt da sie manchmal die einzige Möglichkeit darstellen, eine Berechnung auszudrücken und normalerweise mit Hilfe von Zeigern kompakter und effizienter programmiert werden kann.

Zeiger und Adressen

Da ein Zeiger die Adresse eines Objektes enthält, kann man über den Zeiger auf indirektem Wege auf das Objekt zugreifen. Dies läßt sich am einfachsten anhand eines Beispiels erläutern:
Nehmen wir an, x und y seien Variablen, beispielsweise vom Typ int und p_x ein Zeiger, der auf eine noch nicht näher angegebene Art und Weise definiert wurde. Die Anweisung

```
p_x = &x;
```

weist dem Zeiger p_x die Adresse von x zu. Dabei liefert der Adressoperator & die Adresse. px zeigt also jetzt auf x. Um nun indirekt über den Zeiger auf die variable x zuzugreifen, und ihren Wert beispielsweise der Variablen y zuzuweisen, bedient man sich folgender Anweisung:

```
y = *p_x;
```

Der sog. Verweis-Operator * behandelt seinen Operanden als die Adresse des eigentlichen zielobjektes, und benützt diese Adresse, um den Wert der des Zielobjektes zu manipulieren. * ist quasi das Gegenstück zu &.
Wie werden Zeiger nun definiert? Die Definition eines Zeigers ähnelt der einer beliebigen Variablen.

```
int *p_x;
```

definiert den Zeiger für das obige Beispiel. Es festgelegt, das es sich bei p_x um einen Zeiger auf eine Variable des Typs `int` handelt. Das hat zur Folge, daß es nicht erlaubt ist, p_x die Adresse einer Variable beispielsweise vom Typ `double` zuzuweisen. Es hat sich als nützlich erwiesen, ein p oder p_ vor den Namen des Zeigers zu setzen, um von vornherein klar zu machen, daß es sich bei der Variable um einen Zeiger handelt (Pointer = engl. Zeiger). Dies beugt Fehlern beim Umgang mit Zeigern vor, einer der häufigsten Fehlerursachen beim Programmieren in C.
Zeiger lassen sich dementsprechend auch in Ausdrücken verwenden.

```
y = *p_x + 1;
```

Diese Anweisung beispielsweise addiert den Wert eins zum Inhalt der Variablen x und weist das Ergebnis y zu. Analog weist die folgende Anweisung x den Wert von y + 1 zu:

```
*p_x = y + 1;
```

Zeiger und Arrays
In C hängen Zeiger und Arrays sehr eng zusammen. Jede Operation mit Array-Indizes kann auch mit Zeigern ausgedrückt werden.
Die Anweisung

```
int a[6];
```

deklariert ein Array mit 6 Elementen, numeriert von a[0] bis a[5].
Deklariert man p_a als Zeiger auf einen int-Wert

```
int *p_a;
```

so kann man nach der Zuweisung

```
p_a = &a[0];
```

über den Zeiger p_a auf das Array zugreifen. p_a zeigt dann auf das Element a[0],

```
x = *p_a;
```

hat demnach das gleiche Ergebnis wie die Anweisung

```
x = a[0];
```

Zeigt p_a auf ein Element des Arrays, so zeigt p_a + 1 auf das nachfolgende Element, d.h. durch erhöhen des Wertes von p_a lassen sich die einzelnen Array-Elemente durchlaufen. Zeigt also p_a auf a[0] so liefert

```
*(p_a + 1)
```

den Inhalt von a[1].
Dies gilt unabhängig vom Datentyp oder der Größe der Elemente im Array a. Die Bedeutung von "addiere 1 zu einem Zeiger", und als Verallgemeinerung die gesamte Arithnetik mit Zeigern, ist so definiert, daß p_a + 1 auf das nächste Objekt zeigt und daß pa + i auf das i-te Objekt nach p_a zeigt.
Es besteht also ein enger Zusammenhang zwischen Array-Indizes und Zeigerarithmetik. Nach Sprachdeinition ist der Wert einer Variablen oder eines Ausdrucks vom Typ Array die Adresse des Elementes 0 (des Anfangselementes) des Array. Deshalb sind nach der Zuweisung

```
p_a = &a[0];
```

die Werte von p_a und a identisch. Da der Name eines Vektors synonym zur Adresse des Anfangselements ist, kann man die Zuweisung pa = &a[0] auch so formulieren:

```
pa = a;
```

Wesentlich überraschender ist die Tatsache, daß statt a[i] auch *(a+i) geschrieben werden kann. Der Compiler wandelt a[i] sofort in *(a+i) um; die zwei Angaben sind äquivalent. Einen Unterschied zwischen Arraynamen und Zeiger muß man sich allerdings merken: Ein Zeiger ist eine Variable, also sind pa = a und pa++ erlaubt. Ein Array-Name ist jedoch keine Variable: Ausdrücke wie a = pa oder a++ sind nicht erlaubt.

Speicherverwaltung mit Zeigern

Wie bereits angedeutet wurde, kommt es bei Programmieraufgaben oft vor, daß bei der Programmerstellung Umfang und Art der zu bearbeitetenden und zu speichernden Daten nicht festgelegt ist. Für derartige, häufig auftretende Fälle eignet sich die sogenannte dynamische Speicherverwaltung. Mit Hilfe der Standardbibliotheksfunktion `malloc` lassen sich in einem Teil des Hauptspeichers, dem sogenannten Heap (engl. Halde) Speicherblöcke definierter Größe nach Bedarf reservieren, der dann nach Gebrauch mit Hilfe der `free`-Funktion wieder freigeben werden muß. Zum Zugriff auf den reservierten Speicherblock bedient man sich eines Zeigers.
Definiert man zum Beispiel den Zeiger p mit

```
char *p;
```

als Zeiger auf eine Variable vom Typ `char`, und blockgroesse mit

```
int blockgroesse = 10;
```

so kann mit Hilfe der Anweisung

```
p = (char *) malloc(blockgroesse);
```

ein Speicherblock mit der durch blockgroesse festgelegten Anzahl char-Variablen angefordert werden. Kann `malloc` erfolgreich ausgeführt werden, d.h. es ist noch hinreichend viel Speicher verfügbar, so liefert die Funktion einen Zeiger auf den Beginn des reservierten Speicherblocks zurück. War `malloc` nicht erfolgreich, so erhält p den Wert `NULL`.
Vor einem Zugriff auf den reservierten Speicher sollte unbedingt überprüft werden, ob `malloc` erfolgreich war, da es sonst beim Zugriff auf den "inWirklichkeit" nicht reservierten Speicher zu einem Programmabsturz kommen kann. Das gilt auch für das Freigeben des Speichers mit `free`.
Dies erledigt zum Beispiel folgende Anweisungsfolge:

```
#include <stdlib.h>
```

```
#include <malloc.h>
#include <string.h>

typedef int BOOL;
#defineFALSE 0
#define TRUE 1

void main()
{
  char *p;
  int nBlockgroesse = 255;

  p = (char*) malloc (nBlockgroesse);
  if (p != NULL){
       strcpy(p, "Dies ist ein einfacher
Speicherblock");
       free (p);
  }
}
```

Es ist von großer Wichtigkeit, den erhaltenen Speicher nach Gebrauch wieder freizugeben. Wird dem Zeiger, der auf den Speicherblock verweist (im Beispiel p) vor der Freigabe des Speichers durch `free` ein anderer Wert zugewiesen, so ist dieser Speicherblock quasi "verloren", d.h er kann im weiteren nicht mehr benutzt werden.
Mit Hilfe dieser Mechanismen lassen sich auch wesentlich komplexere Datenstrukturen dynamisch erzeugen. Auf diese wird in Kapitel 0 noch näher eingegangen.

Funktionen

Durch die Verwendung von Funktionen können große Problemstellungen in kleinere Teilprobleme untergliedert werden, wodurch sich eine wesentlich übersichtlichere Programmstruktur ergibt. Dabei gilt wie für auch für Variablen, daß jede Funktion bevor sie in einer C-Quellcode-Datei genutzt werden kann, zunächst bekannt gemacht (deklariert) werden muß. Dies geschieht durch die Definition eines sogenannten „Prototyps“, der nichts anderes ist, als ein Aufrufmuster für die entsprechende Funktion. Die Syntax bei der Definition von Funktionen ist dabei wie folgt:

```
<Rückgabetyp> <Funktionsname> (<Übergabetyp>[,
(<Übergabetyp2>[, (<Übergabetyp3> ...]]);
```

Das folgende Beispiel deklariert die Funktion `Stringausgeben(..)`.

Beispiel:

```
int Stringausgeben(const char*, const int);
```

Hierbei wird ein Zeiger auf den auszugebenden String übergeben und ein Integer, der zum Beispiel angeben kann, wie viele Zeichen des Strings ausgegeben werden müssen. Der Typ `int`, der vor dem Funktionsnamen steht beschreibt den Typ des Rückgabewertes, den die Funktion über `return()` an die Aufruffunktion (z.B. `main`) zurückgibt. Die Deklarationszeile muß, wie bereits erwähnt, in jeder Datei eingefügt werden, in der die Funktion genutzt wird. Die Implementierung hingegen darf nur einmal vorgenommen werden. Sie könnte für `Stringausgeben` wie folgt aussehen:

Beispiel:

```
int Stringausgeben(const char * lpctstrString, int
nNumberOfCharacters)
{
  int nCounter = 0;
  do{
       if(lpctstrString[nCounter] != '\0')
            putchar(lpctstrString[nCounter]);
       else
            return -1;
       nCounter++;
  }while(nCounter < nNumberOfCharacters);
  return 0;
}
```

Um selbst implementierte Funktionen in beliebig vielen Quelldateien nutzen zu können, wird dabei in aller Regel die in Bild 0.1 dargestellte Filestruktur verwendet.

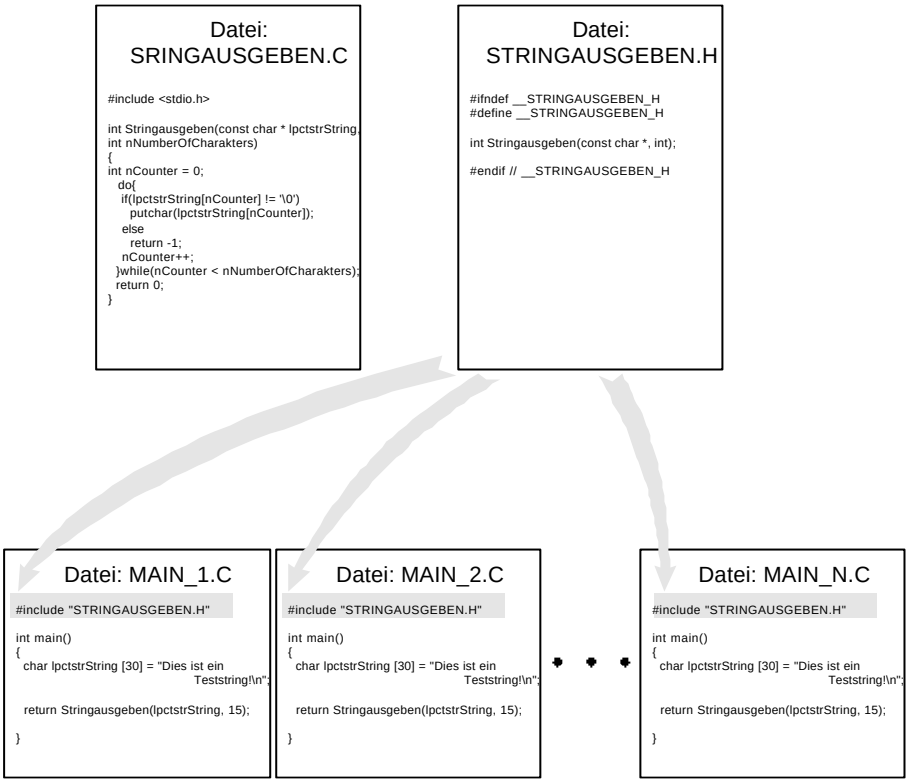


Bild 0.1: Einbinden von selbst deklarierten Funktionen.

Benutzerdefinierte Datenstrukturen, Structs, Unions

In diesem Kapitel geht es hauptsächlich um komplexere Datenstrukturen, die im Gegensatz zu den in Kapitel 0 behandelten einfachen Datentypen dazu in der Lage sind, auch umfangreichere Datenansammlungen unter einem Bezeichner abzuspeichern und den Zugriff auf einzelne Teile dieser Daten zu ermöglichen. Die beiden wesentlichen, komplexeren Datenstrukturen, die C bietet sind `struct` und `union`.

"Structs"

Ein "struct" ist die Zusammenfassung mehrerer einzelner Variablen zu einem Verbund, der später unter einem Namen angesprochen werden kann. Im folgenden wird ein einfaches Beispiel für die Deklaration eines "structs" vorgestellt, das die drei Koordinaten eines dreidimensionalen Punktes enthalten soll, wie er etwa im CAD-Bereich oft benötigt wird:

```
struct Point3D
{
 float x;
 float y;
 float z;
 long number;
}
```

Man sieht an diesem Beispiel, daß die Struktur Point3D dazu dient, drei gleichartige Variablen vom Typ `double` aufzunehmen, wobei jede Koordinate (x, y und z) durch je eine Variable beschrieben wird. Zusätzlich enthält die Struktur in diesem Beispiel noch eine Integer-Variable, um jeden Punkt mit einer eindeutigen Nummer versehen zu können.
Allgemein lassen sich innerhalb eines "structs" beliebig viele Variablen zusammenfassen, wobei natürlich irgendwann die Grenzen der Übersichtlichkeit erreicht werden. Die allgemeine Form der Deklaration sieht wie folgt aus:

```
struct NAME {
 TYP1 VAR_NAME1;
 TYP2 VAR_NAME2;
 ... }
```

Das Ergebnis entspricht dann dem im Bild 0.1 dargestelltem:

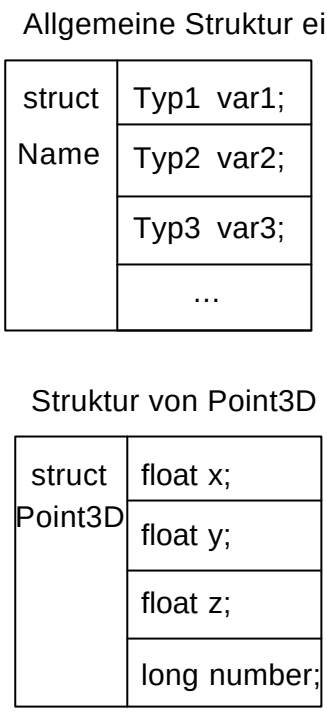


Bild 0.1: Aufbau und Funktionsweise eines structs.

Im folgenden ist ein kleines Hauptprogramm angegeben, in dem zwei Variablen vom Point3D definiert werden. Nach dem Einlesen der Werte für alle drei Koordinaten beider Punkte erfolgt eine komponentenweise Addition (Vektoraddition) und das Ergebnis erscheint als Ausgabe auf dem Bildschirm. In diesem Beispiel erfolgt die Deklaration der Struktur in einem Headerfile, und es wird ihr direkt ein Name ("Point3D") zugeordnet. Anschließend kann man an allen Stellen im Programm, an denen die Struktur bekannt ist, weitere Variablen vom Typ "Point3D" deklarieren. Diese Vorgehensweise wird des öfteren verfolgt, da es häufig vorkommt, daß neu deklarierte Strukturen auch in anderen Bereichen im Programm verwendet werden sollen. Man beachte, daß der Definition der Variablen unbedingt das Schlüsselwort `struct` voranzustellen ist, da der Compiler sonst eine Fehlermeldung ausgibt (Ausnahme: Definition mit `typedef` s.u.).

```
#include "stdlib.h"
#include "point3d.h"

main()
{
 float in; /* Variable zur Eingabe der Koordinaten
*/
 struct Point3D a,b;

 clrscr(); /* Bildschirm löschen */
 printf("Bitte die x-Koordinate des 1. Punktes
eingeben\n");
```

```
 scanf("%f",&in);
 a.x = in;
 printf("Bitte die y-Koordinate des 1. Punktes
eingeben\n");
 scanf("%f",&in);
 a.y = in;
 printf("Bitte die z-Koordinate des 1. Punktes
eingeben\n");
 scanf("%f",&in);
 a.z = in;
 a.number = 1;
 printf("Bitte die x-Koordinate des 2. Punktes
eingeben\n");
 scanf("%f",&in);
 b.x = in;
 printf("Bitte die y-Koordinate des 2. Punktes
eingeben\n");
 scanf("%f",&in);
 b.y = in;
 printf("Bitte die z-Koordinate des 2. Punktes
eingeben\n");
 scanf("%f",&in);
 b.z = in;
 b.number = 2;
 printf ("Summe der x-Komponenten:%f\n",a.x+b.x);
 printf ("Summe der y-Komponenten:%f\n",a.y+b.y);
 printf ("Summe der z-Komponenten:%f\n",a.z+b.z);
 exit(0);
}
```

Man kann das "struct" auch direkt im Hauptprogramm deklarieren und Variablen von ihm erzeugen. Dann muß man die Zeile:

```
struct Point3D a,b; /* Variablen des Typs Point3D,
die jeweils einen Punkt enthalten */
```

durch folgende Deklaration ersetzen:

```
struct
  {
   float x;
   float y;
   float z;
   long number;
  }a,b;  /* Variablen des Typs Point3D, die jeweils
einen Punkt enthalten */
```

und außerdem wird die `#include`-Anweisung in der zweiten Zeile nicht mehr benötigt.
Wird die Deklaration so durchgeführt, ist es nicht mehr möglich, an anderer Stelle erneut Variablen von diesem `struct` zu erzeugen, da es keinen Namen erhalten hat und somit nicht mehr zugreifbar ist.

Zugriff auf Komponenten

Der Zugriff auf die einzelnen Komponenten eines "structs" erfolgt mit der sogenannten "Punktnotation", wie sie auch im Hauptprogramm verwendet wird:

```
a.x = ein; /* Schreibender Zugriff auf die
Komponente x der Struktur a */
ein = a.x; /* Lesender Zugriff auf die Komponente x
der Strukutur a */
```

Werden Strukturvariablen über Zeiger angesprochen, so wird statt der Punktnotation die Pfeilnotation verwendet (s.u.), wobei der lesende und schreibende Zugriff auf die Variable a aus der Struktur x dann wie folgt aussieht:

```
a->x = ein; /* Schreibender Zugriff auf die
Komponente x der Struktur a über Pointer */
ein = a->x; /* Lesender Zugriff auf die Komponente x
der Strukutur a über Pointer */
```

Näheres zu diesem Vorgehen s.u. und in folgendem Bild 0.2:

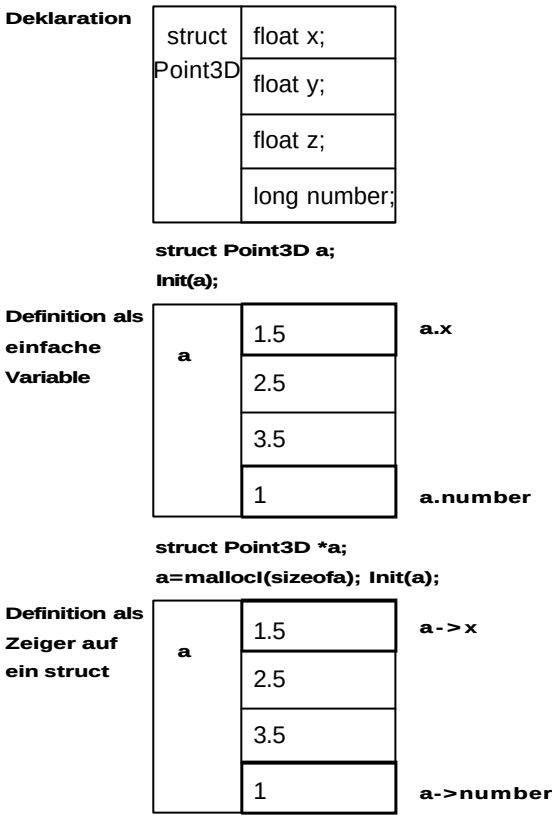


Bild 0.2: Zugriff auf die Komponenten eines "structs".

Neue Typnamen erzeugen mit "typedef"

Mit Hilfe von Strukturen ist es z.B. auch möglich, mehr als einen Funktionswert aus einer Funktion zurückzugeben, indem man alle "einfachen" Werte, die man zurückgeben möchte in einem `struct` zusammenfaßt und dieses dann als einzigen Wert aus der Funktion zurückgibt. Das setzt natürlich voraus, daß die Funktion den Rückgabetyp der Struktur besitzt. Um einen solchen Typen zu erzeugen, benutzt man das Schlüsselwort `typedef` vor einer `struct`-Definition, wie im folgenden Beispiel:

Header-Datei mit Deklaration des "structs":

```
typedef struct
{
 int x;
 int y;
}Ret_stru;
```

Hauptprogramm zum Testen:

```
#include "stdlib.h"
#include "ret_stru.h"
```

```
/* Funktionsdeklaration */
Ret_stru test();

/* Hauptprogramm */
main()
{
 Ret_stru main_value; /*Anlegen einer Variablen
main_value vom Type Ret_stru */

 /*Initialisierug der Variable*/
 main_value.x = 0;
 main_value.y = 0;
 main_value = test(); /* Aufrufen der Funktion */
 /* Bildschirmausgabe */
 printf("main_value.x:%d\n",main_value.x);
 printf("main_Value.y:%d\n",main_value.y);
 exit (0);
}

/* Funktionsimplementierung */
Ret_stru test()
{
 Ret_stru value;
 /* Zuweisen von 2 Werten an die Variablen des
structs */
 value.x = 1;
 value.y = 2;
 /* Rückgabe des structs */
 return value;
}
```

In obigem Beispiel wird das Schlüsselwort `typedef` verwendet um einen neuen Datentypnamen, nämlich `Ret_stru` zu erzeugen. Anschließend kann man Funktionen definieren, die diesen Typen als Rückgabetypen (s. Kapitel 0) besitzen. Der Vorteil dieses Vorgehens ist der, daß der neue Datentypname

im folgenden Programm genauso verwendet werden kann, wie die bereits vorhanden Datentypen (z.B. `int` oder `float`). Man beachte außerdem die etwas andere Syntax der `struct`-Deklaration bei der Verwendung von `typedef`.

Implementierung einer einfachen linearen Liste

Eine weitere Besonderheit von Strukturen in C ist die, daß man bereits innerhalb der Deklaration einer Struktur, rekursiv genau auf diese Struktur zurückgreifen kann. Häufig wird dieses Vorgehen zum Aufbau dynamischer Speicherstrukturen im Rechner verwendet. Da dies ein typisches Beispiel für den Einsatz von "structs" ist, soll an dieser Stelle noch einmal auf die in Kapitel 0 gelegten Grundlagen zur dynamischen Speicherverwaltung zurückgegriffen werden, und der Aufbau und Durchlauf ein linearen Liste programmiert werden. Die einzelnen Listenelemente beinhalten aus Gründen der Übersichtlichkeit lediglich eine Integerzahl. Ein Listenelement besteht damit aus einem Zeiger auf seinen Nachfolger und der erwähnten Integerzahl. Der Aufbau der Liste ist in folgendem Bild beschrieben.

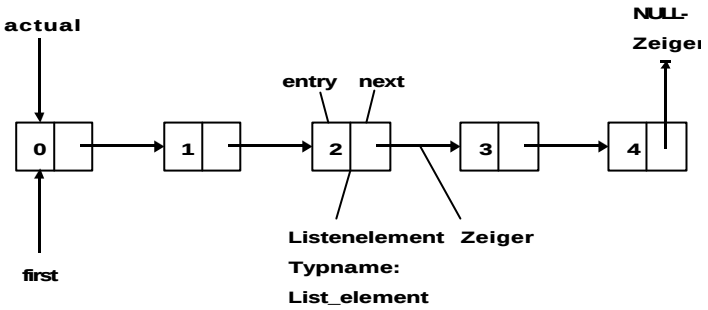


Bild 0.3: Struktur einer einfachen linearen Liste.

Im Programm werden zunächst zwei Zeiger definiert, die auf den Typ "List_element" zeigen können. Diese werden initial direkt auf den "NULL-Pointer" gesetzt. Dann beginnt der Aufbau der Liste, indem ein Startelement des Typs "List_element" mit Hilfe der "`malloc`"-Funktion erzeugt wird. In der sich anschließenden "`for`"-Schleife werden vier weitere Listeneinträge an das erste angehängt. Die Vorgehensweise zum Anhängen kann man dabei dem folgendem Bild 0.4 entnehmen.

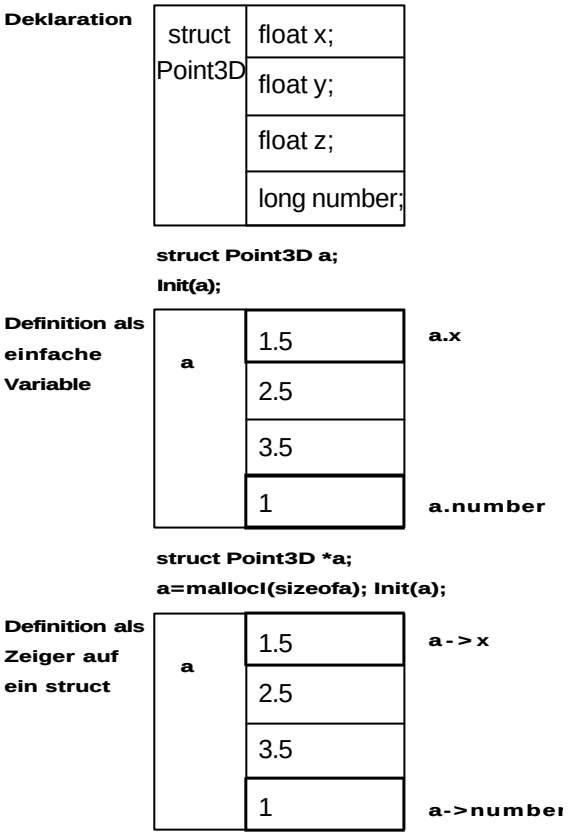


Bild 0.4: Aufbau einer linearen Liste mit fünf Elementen.

Dann wird der aktuelle Zeiger, der jetzt auf dem letzten Listenelement steht wieder auf das erste zurückgesetzt und der Durchlauf der Liste zur Ausgabe auf dem Bildschirm beginnt (vgl. folgenden Sourcecode).

```
#include "stdlib.h"
#include "list_ele.h"

/* Hauptprogramm */
main()
{
 int i;
 List_element *first = NULL; /* Zeiger auf den Start
der Liste */
 List_element *actual = NULL; /* Zeiger auf das
aktuelle Listenelement */

 /* 1. Listenelement erzeugen und Zeiger
Initialisieren */
```

```
 first =
(List_element*)malloc(sizeof(List_element));
 first->entry = 0;
 first->next = NULL;
 actual = first;
 /* 4 weitere Listenelemente erzeugen und hinten an
die Liste hängen */
 for(i=1;i<=4;i++)
 {
  actual->next =
(List_element*)malloc(sizeof(List_element));
  actual = actual->next;
  actual->entry = i;
  actual->next = NULL;
 }
 actual = first; /* Zeiger wieder auf Anfang */
 /* Liste komplett ausgeben */
 do
 {
  printf("Listeneintrag:%d\n",actual->entry);
  actual = actual->next;
 }
 while (actual != NULL);
 exit(0);
}
```

Im Zusammenhang mit der oben erwähnten rekursiven Deklarationsweise ist die Deklaration von "List_element" im Headerfile von Interesse. Sie ist hier angegeben:

```
typedef struct List_element
{
 struct List_element *next;
 int entry;
} List_element;
```

Wie man sieht, bezieht sich bereits die Deklaration des "structs" auf sich selbst. Das Nachfolgeelement soll wieder ein Zeiger auf ein Listenelement sein. Eine solche Deklaration ist in C erlaubt und durchaus üblich. Es gäbe auch gar keine andere Möglichkeit, eine Komponente einer Struktur auf die gleiche

Struktur zeigen zu lassen, respektive sie dafür vorzubereiten. Man beachte, daß gleichzeitig mit der Deklaration der Struktur auch ein neuer Typname "List_element" vereinbart wird, der im Hauptprogramm genauso verwendet werden kann, wie ein "eingebauter" Typname.
Das Listenbeispiel zeigt eine ganz typische Anwendung für "structs", wie sie in C häufig eingesetzt wird. Der Einsatz von Strukturen erhöht die Übersichtlichkeit in der Programmierung und ist außerdem die Basis für die spätere Erweiterung von C um Klassen und Objekte (s. C++ Teil dieses Skriptes).

"Unions"

Eine weitere Möglichkeit komplexere Datenstrukturen zu bearbeiten, bietet das `union`- Konstrukt. Bei seiner Benutzung wird fast genauso vorgegangen wie bei der eines "structs", nur das nicht für alle Komponenten Speicherplatz bei der Defintion einer Variablen vergeben wird, sondern nur für eine einzige. D.h. es kann zu einem Zeitpunkt genau eine Komponente aus der Anzahl der deklarierten Komponenten ausgewählt und benutzt werden. Bei Auswahl einer anderen Komponente wird die zuerst benutzte Komponente überschrieben. Die Größe des zu reservierenden Speicherbereichs richtet sich nach der größten, in der `union` vorgesehenen Komponente. Allgemein ist die Syntax zur Definition einer `union` die folgende:

```
union NAME {
 TYP1 VAR_NAME1;      /* entweder Benutzung dieser
Komponente */
 TYP2 VAR_NAME2;      /* oder dieser */
 ... }
```

Das `union`-Konstrukt wird häufig dann verwendet, wenn von vornherein feststeht, daß nur eine Komponente der Struktur benutzt wird. Es gibt z.B. die Möglichkeit in der oben vorgestellten Struktur "Point3D" die Koordinaten entweder als ganzzahligen Wert oder als Fließkommazahl abzulegen. Ist bekannt, daß der 3D-Punkt immer nur in einer von beiden Varianten benutzt werden soll, kann die `union` zum Einsatz kommen. Zur Einfachheit wird im Hauptprogramm nur die x-Koordinate eingelesen und ausgegeben; mit den anderen Koordinaten kann analog verfahren werden. Zunächst die Deklaration im Headerfile:

```
struct Point3D
{
 union
 {
       float fx;
       int ix;
 }x_koord;
 union
 {
        float fy;
        int iy;
 }y_koord;
 union
 {
```

```
      float fz;
      int iz;
 }z_koord;
 long number;
}
```

Nun das Hauptprogramm:

```
#include "stdlib.h"
#include "pun3d.h"

main()
{
 float fin; /* Variable zur Eingabe der Koordinaten
als Fließkommazahl */
 int iin; /* Variable zur Eingabe der Koordinaten
als ganze Zahl */
 int choice; /* Auswahlvariable */
 struct Point3D a; /* Definition einer Variablen des
Typs Point3D */

 clrscr(); /* Bildschirm löschen */
 printf("Soll die Koordinate als ganze Zahl (1
Eingeben) oder als Fließkommazahl (2 Eingeben)
angegeben werden?\n");
 scanf("%d",&choice);
 switch (choice)
 {
  case 1:
  {
     printf("Bitte die x-Koordinate des 1. Punktes
eingeben\n");
     scanf("%f",&fin);
     a.x_koord.fx = fin;      /* Zuweisung an die
Fießkommakomponente */
```

```
    }break;
    case 2:
    {
      printf("Bitte die x-Koordinate des 1. Punktes
eingeben\n");
      scanf("%d",&iin);
      a.x_koord.ix = iin;       /* Zuweisung an die
Integerkomponente */
    }break;
    default:
    break;
   }
   printf ("Fließkommakomponente
x:%f\n",a.x_koord.fx);
   printf ("Integerkomponente x:%d\n",a.x_koord.ix);
   exit(0);
  }
```

Im vorangegangenen Beispiel erkennt man die Deklaration und Benutzung einer Datenstruktur, die innerhalb eines "structs" mehrere "unions" verwendet, um die wahlweise Speicherung der Punktkoordinaten in Ganz- oder Fließkommazahlen zu ermöglichen.
"unions" dienen i.w. dazu Speicherplatz zu sparen und sollten immer dann verwendet werden, wenn man eindeutig festlegen kann, daß eine Komponente einer Struktur wahlweise den einen oder den anderen Typ zu einem Zeitpunkt hat, nie aber beide. Nach Eingabe und Übersetzung des Beispielprogramms sieht man, daß je nach Auswahl des Datenformats eine von beiden Komponenten undefinierte Werte aufweist, während die andere den eingegebenen Wert enthält.
Literatur zu "structs"; "unions" und Listen findet sich in [7][8].

Literatur zu der Programmiersprache C

[1] N.N *Ohne Titel*
Begleitunterlagen zum „Hardwarepraktikum für Ingenieurinformatiker“,
FB I Informatik, Universität Dortmund

[2] N.N. *Online-Hilfe zum Microsoft Developer Studio `97*
Microsoft Corporation

[3] N.N. *Softwaretechnologie*
Skriptum zur GlechnamigenVorlesung
FB Informatik, LS X der Universität Dortmund

[4] N.N. *Online Hilfe zu Borland Delphi 3*
Borland, 1996

[5] N.N. *Turbo C++ 3.0 Benutzerhandbuch*
Borland, 1992

[6] Schildt, H. *Turbo C/C++ - the Complete Reference*
Osborne McGraw-Hill,
Berkeley, 1990

[7] Kernighan, B. W.
Ritchie, D. M. *Programmieren in C*
Institut für Spanende Fertigung, Universität Dortmund
Unveröffentlichte Studienarbeit, 1998

[8] Weinert, K.
(Schreiner, A.T.
Janich, E.
beide Hrsg.) *Spanende Fertigung III*
Reihe: PC-Professionell
Carl Hanser Verlag,
München Wien, 1983

[9] Precht, M.
Meier,N.
Kleinlein J. *EDV-Grundwissen - Eine Einführung in Theorie und Praxis der modernen EDV*
4. Auflage,
Addison-Wesley-Longmann,
Bonn, 1997

Klassen und Objekte

Zu Beginn des C++-Teils des Skriptes, wird dieses Kapitel die beiden zentralen Elemente von C++: "Klassen" und "Objekte" einführen. Sie stellen gleichsam die wesentlichste Erweiterung der Sprache C dar, um Objektorientierung zu ermöglichen. Es ist von großer Wichtigkeit für das weitere Verständnis gleich zu Beginn die beiden Begriffe sauber zu definieren und abzugrenzen, so daß zunächst eine abstrakte, allgemeine Beschreibung erfolgt, bevor im Anschluß daran die programmtechnische Umsetzung beschrieben wird.
Eine Klasse stellt die Beschreibung eines abstrakten Datentypen dar, d.h. sie ermöglicht die Definition benutzerdefinierter Datentypen, ebenso wie die Erweiterung bereits vorhandener.
Ein Objekt ist eine konkrete, zur Laufzeit des Programms definierte Variable eines solchen abstrakten Datentyps. Analog dazu kann eine Variable vom Typ "int" als Objekt der Klasse "int" betrachtet werden und eine solche Variable ist in C++ auch nichts anderes. Dieser Zusammenhang ist von zentraler Bedeutung für das Verständnis von C++ und anderen objektorientierten Programmiersprachen. Eine Klasse beschreibt zur Entwurfszeit des Programms die Eigenschaften (den "Status") und die Fähigkeiten (die "Methoden"), der zur Laufzeit von ihr abgeleiteten Objekte. Objekte werden also nicht "in" einer Klasse definiert sondern "durch" eine Klasse. Es können beliebig viele Objekte einer bestimmten Klasse erzeugt werden, aber jedes erzeugte Objekt gehört zu genau einer Klasse. Objekte können zur Laufzeit miteinander kommunizieren, sie können Objekte aus anderen Klassen enthalten und sie können aufgrund der in ihrer Klasse beschriebenen Methoden Aktionen durchführen. Außerdem können weitere Beziehungen (z.B. Vererbung) zwischen Objekten bestehen, wobei diese Beziehungen ebenfalls durch Klassendeklarationen beschrieben werden. Zur Verdeutlichung des Verhältnisses von Klassen und Objekten dient auch das folgende Bild:

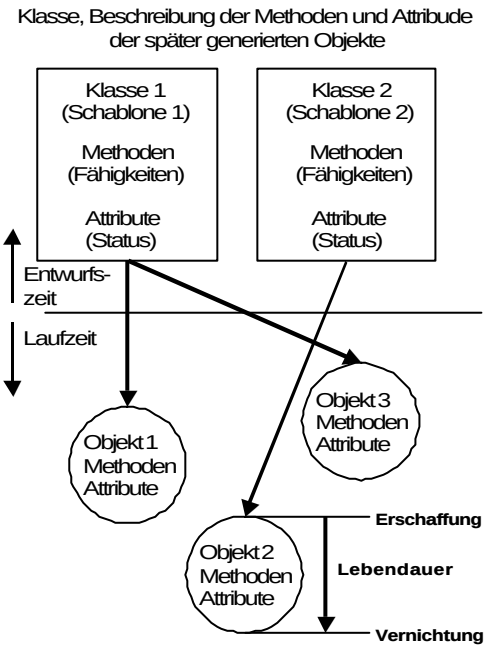


Bild: Verhältnis von Klassen und Objekten.

Beispiel: Bankkonto

Das generelle Vorgehen in der objektorientierten Programmierung sieht immer so aus, daß man versucht die Problemstellung durch eine Ansammlung von Objekten und die Beziehungen zwischen ihnen zu beschreiben. Die Objekte müssen dann einzelnen Klassen zugeordnet werden, welche anschließend zu implementieren sind.

Zur Veranschaulichung wird im folgenden, anhand des einfachen Beispiels eines Bankkontos, eine Klasse entworfen. Anschließend werden einige Objekte dieser Klasse erzeugt und manipuliert. Der Status des Bankkontos soll vereinfachend durch seine Kontonummer und seinen aktuellen Kontostand beschrieben sein. Zusätzlich hat ein Kontoobjekt die Fähigkeit seinen Kontostand zu ändern, d.h. Geld abzuheben oder einzuzahlen. Abgesehen davon muß sich ein Kontoobjekt auf einen definierten Startzustand bringen (Konto anlegen) und gegebenenfalls auch wieder korrekt löschen können (Konto kündigen). Man sieht an dieser Beschreibung bereits, daß in der objektorientierten Programmierung immer vom Objekt ausgegangen wird, dem bestimmte Eigenschaften und Fähigkeiten zugeschrieben werden. In einer fiktiven, umgangssprachlichen Klassendeklaration, die eine Anzahl von Kontoobjekten beschreibt, sähe das etwa so aus:

```
Objekte der Klasse "Konto" bestehen aus:
Statusbeschreibung:
Kontonummer (Lange Ganze Zahl)
Aktueller Kontostand (Reelle Zahl)
Fähigkeiten:
Konto initialisieren (Parameter: Betrag (Reelle
Zahl))
Konto löschen (Parameter: -)
Geld einzahlen (Parameter: Betrag (Reelle Zahl))
Geld abheben (Parameter: Betrag (Reelle Zahl))
Kontonummer ausgeben (Parameter: Kontonummer (Lange
Ganze Zahl)
Ende der Deklaration
```

Man erkennt bereits an dieser Beschreibung einige wesentliche Eigenschaften einer Klassendeklaration. Die Beschreibung des Status, bzw der "Attribute" durch sogenannte "Member-Variablen" erfolgt durch Variablendeklarationen innerhalb der Klassendeklaration.
Die Fähigkeiten der Objekte einer Klasse werden durch Funktionsdeklarationen innerhalb der Klassendeklaration festgelegt, wobei diese Funktionen "Member-Funktionen" oder "Methoden" genannt werden. Genau hier liegt die zentrale Erweiterung zu den "structs" in C. "Structs" können lediglich Variablen verschiedener Typen enthalten (vgl. Kapitel 0 des C-Kurses) und fassen diese zusammen. In diesem Zusammenhangt ist der Begriff der Kapselung ebenfalls sehr wichtig. Er beschreibt das "Verschwinden" von internen Informationen in einer Klasse, ein Konzept das als "Information Hiding" bekannt ist und auf das weiter unten erneut eingegangen wird (vgl. nächstes Kapitel).

Eine Klasse in C++

Kommen wir zur programmtechnischen Umsetzung dieser Konzepte in C++. Das Programm besteht aus drei Sourcefiles, wobei zwei die Endung ".cpp" tragen und eines die Endung ".h" trägt. Im File "konto.h" befindet sich die Klassendeklaration (auch Schnittstellen-, Interface- oder Headerdatei genannt). Sie sieht folgendermaßen aus:

```
// Headerfile zur Klasse Konto
#ifndef _KONTO_H_
#define _KONTO_H_
```

```
class Konto
{
 // Öffentliche Daten und Funktionen
 public:
        // Member-Variablen
        long m_lKontonummer;
        // Methoden
        Konto(long,double);  // Konstruktor
        ~Konto();                  // Destruktor
         void einzahlen(double);
         void abheben(double);
         void kontostand_ausgeben();
 // Private Daten und Funktionen
 private:
         double m_dKontostand;
};
#endif
```

Zunächst einmal sorgen die beiden Präprozessor-Anweisungen (s. Kapitel 0 des C-Skriptes) dafür, daß die Datei nicht öfter als einmal in eine ".cpp"-Datei eingebunden werden kann, um multiple Deklarationen zu verhindern. Wurde die ".h"-Datei noch nicht in eine bestimmte ".cpp"-Datei eingebunden, so wird das "Makrolabel" `_KONTO_H_` definiert. Beim nächsten Versuch konto.h in die gleiche ".cpp"-Datei einzubinden, wird die gesamte Klassendeklaration einfach übersprungen, da sie komplett zwischen den Präprozessoranweisungen `#ifndef` und `#endif` steht. Solche mehrfachen Einbindungen können in größeren Softwareprojekte auftreten, wenn nämlich Headerdateien in andere Headerdateien eingebunden werden müssen, was man an sich zu vermeiden sollte, was aber manchmal nicht zu verhindern ist.
Als nächstes folgt das Schlüsselwort `class` und schließlich der Klassenname: "Konto" (Klassenbezeichner haben oft auch eigene Präfixe, z.B. CKonto o.ä., wir halten uns in diesem Skript lediglich an die Konvention den ersten Buchstaben eines Klassennamens groß zu schreiben). Nach den geschweiften Klammern folgt das C++ Schlüsselwort `public`:. Dieses Schlüsselwort gibt an, daß die folgenden Deklarationen "öffentlich", d.h. von außerhalb der Klasse aus zugreifbar sind. Näheres zu den einzelnen Sicherheitsstufen folgt im nächsten Kapitel. In der nächsten Zeile wird die öffentliche "Member-Variable" "m_lKontonummer" deklariert und im Anschluß daran die Methoden, die ebenfalls "public" sind. Schließlich gibt es noch einen "private"-Teil in dieser Klassendeklaration, in dem die nicht öffentliche "Member-Variable" "m_dKontostand" deklariert wird. Damit ist die Deklaration der Klasse "Konto" zunächst einmal abgeschlossen und die ".h"-Datei die sogenannte "öffentliche Schnittstelle", oder auch das "Interface" der Klasse. Dieser Teil der Klasse ist im Prinzip das einzige, was ein Benutzer über diese Klasse an Informationen zur Verfügung haben muß, um mit ihr zu arbeiten. Alle anderen Informationen, wie etwa die Datentypen für Kontonummer und Betrag sollen intern in der Implementierung "versteckt" bleiben (s.a. nächstes Kapitel).
Man beachte außerdem, daß die ".h"-Datei nicht in das Projekt eingefügt werden darf, da sie über `#include` eingebunden wird.

Syntax einer Klassendeklaration

```
class NAME
{
  public: (optional)

    // Attribute
    TYPNAME MEMBERVARIABLE;
      ...

    // Methoden
    RÜCKGABETYP FUNKTIONSNAME
                    (PARAMETERLISTE);
     ...

  private: (optional)

    // Attribute
    TYPNAME MEMBERVARIABLE;
      ...

    // Methoden
    RÜCKGABETYP FUNKTIONSNAME
                    (PARAMETERLISTE);
     ...
};
```

Bild: Die prinzipielle Syntax einer Klassendeklaration.

Es fällt an dieser Stelle auf, daß noch nirgends beschrieben wurde wie z.B. das Abheben eines Geldbetrages von unserem Kontoobjekt konkret funktionieren soll. Eigentlich haben wir bislang lediglich Funktionsdeklarationen angegeben, aber diese Funktionen noch nicht mit Leben gefüllt. Genau das geschieht in der Datei "konto.cpp", die die sogenannte "Implementierung" der Klassenmethoden enthält und die wie folgt aussieht:

```
// Implementierung der Klasse Konto
#include "iostream.h" // Einbinden der Standard-
Stream-Klassen
#include "konto.h" // Einbinden der
Klassendeklaration
Konto::Konto(long kontonummer,double startbetrag)
{
 // Initialisierung des Kontos
 m_lKontonummer = kontonummer;
 m_dKontostand = startbetrag;
}
Konto::~Konto()
{
 // Löschen des Kontos
 // An dieser Stelle gibt es erstmal nix zu tun
}
void Konto::einzahlen(double betrag)
```

```
{
 m_dKontostand += betrag;
}
void Konto::abheben(double betrag)
{
 if(m_dKontostand < betrag)
     cout << "Kein Kredit! Transaktion
abgebrochen\n";
 else
     m_dKontostand -= betrag;
}
void Konto::kontostand_ausgeben()
{
 cout << "Kontostand des Kontos: " << m_lKontonummer
<< " beträgt:" << m_dKontostand << "\n";
}
```

In dieser Datei tauchen alle Member-Funktionen aus der Headerdatei wieder auf und werden "implementiert", d.h. ihr Verhalten wird konkret programmiert. Damit der Compiler alles richtig verarbeiten kann, muß als erstes die Headerdatei mit der Klassendeklaration via `#include` eingebunden werden. Dieser Header muß auch an allen anderen Stellen im Programm eingebunden werden, an denen die Klasse Konto benutzt werden soll, um z.B. Objekte dieser Klasse zu erzeugen. Jeder Funktionskopf in der ".cpp"-Datei sieht im Prinzip genauso aus, wie eine normale C-Funktion, nur daß er die zusätzliche Klausel "Klasse::" enthält. Dieser Ausdruck wird benötigt, um dem Compiler mitzuteilen, zu welcher Klasse die entsprechende Member-Funktion gehört, da er das allein durch die `#include`-Präprozessoranweisung nicht ermitteln kann. In unserem Beispiel sagt der Ausdruck "void Konto::einzahlen(double betrag)" also folgendes: Die Funktion "einzahlen()" ist eine Member-Funktion der Klasse "Konto". Sie gibt keinen Rückgabewert zurück ("void") und erhält genau einen Parameter vom Typ "double", nämlich "betrag". In der Schnittstellenbeschreibung der Klasse, ist es nicht notwendig einen konkreten Namen für den Paramter anzugeben, genau, wie bei der Deklaration von Funktionen in C. Im Rumpf der jeweiligen Methode folgt dann der C++-Code, der den Programmablauf dieser Member-Funktion beschreibt. Die Anweisung "cout" ersetzt dabei ab jetzt die Ausgabe mit "printf". Es handelt sich dabei um einen sogenannten "Ausgabestream". Genaueres zu "Streams" findet sich in Kapitel 11 dieses Skriptes. Die Funktionen in der Kontoklasse sind ansonsten sehr simpel gehalten und sollten verständlich sein. Im nächsten Bild ist nochmals die Logik und die Syntax der Implementierung von Member-Funktionen verdeutlicht.

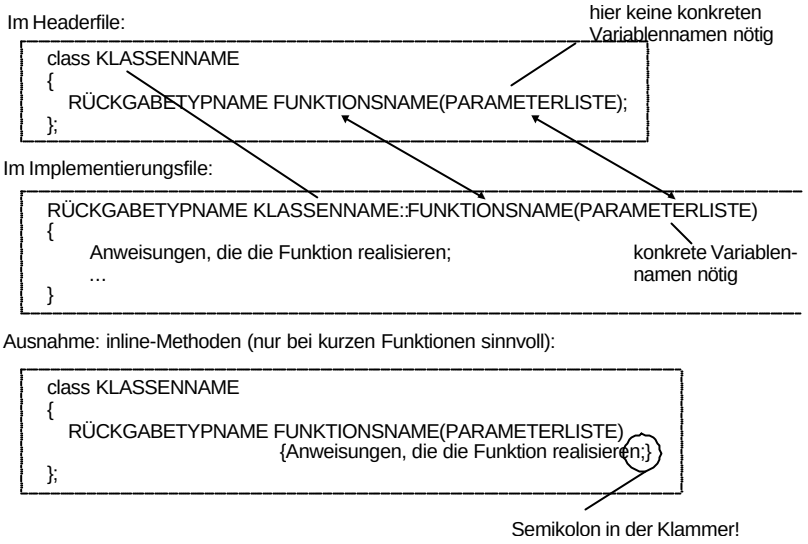


Bild: Logik und Syntax der Methoden-Implementierung.

Erzeugen von Objekten

Neben dem ".h" und ".cpp"-Dateienpaar, die die Klasse Konto zusammen komplett beschreiben, gibt es noch die Datei "bank.cpp" im Projekt. Sie enthält ein einfaches "main()"-Hauptprogramm, daß in C++, genau wie in C obligatorisch ist. Vor Beginn des eigentlichen Hauptprogramms wird auch hier die Datei "konto.h" eingebunden, da im Hauptprogramm Objekte der Klasse "Konto" erzeugt werden sollen. Dieses Erzeugen erfolgt direkt zu Beginn der "main"-Funktion. In den Zeilen:

```
Konto k1(123456,100.0);
Konto k2(221069,50.0);
Konto k3(14443,0.0);
```

, werden drei Objekte der Klasse "Konto" angelegt. Im wesentlichen findet hier nichts anderes als eine Variablendeklaration statt, nur daß der Typ der Variablen ein Abstrakter Datentyp, nämlich die Klasse "Konto" ist. Man spricht in diesem Zusammenhang auch von Instanziierung eines Objektes der Klasse "Konto", bzw. vom Erzeugen von Instanzen einer Klasse. Hier jetzt der Sourcecode des Hauptprogramms:

```
#include "iostream.h"
#include "stdlib.h"
#include "konto.h" // Einfügen der Headerdatei in
bank.cpp

main()
{
 // Erzeugen von 3 Objekten der Klasse Konto
 Konto k1(123456,100.0);
 Konto k2(221069,50.0);
```

```
 Konto k3(14443,0.0);
 // Ausgeben der aktuellen Kontostände der 3 Objekte
 k1.kontostand_ausgeben();
 k2.kontostand_ausgeben();
 k3.kontostand_ausgeben();
 // 50.- von k1 abheben und dann den neuen Stand
ausgeben
 k1.abheben(50.0);
 k1.kontostand_ausgeben();
 // Versuch 100.- von k2 abzuheben (geht nicht, kein
Kredit)
 k2.abheben(100.0);
 // 75.- auf k2 einzahlen und Kontostand ausgeben
 k2.einzahlen(75.0);
 k2.kontostand_ausgeben();
 // Erneuter Versuch 100.- abzuheben (diesmal klappt
es)
  k2.abheben(100.0);
 k2.kontostand_ausgeben();
 // Manipulation und Ausgabe des Standes von k3
 k3.einzahlen(10000.0);
 k3.abheben(5000.0);
 k3.kontostand_ausgeben();
 exit(0); // Programm ordnungsgemäß verlassen
}
```

Im folgenden Bild ist noch einmal das Zusammenspiel der drei Dateien graphisch aufbereitet.

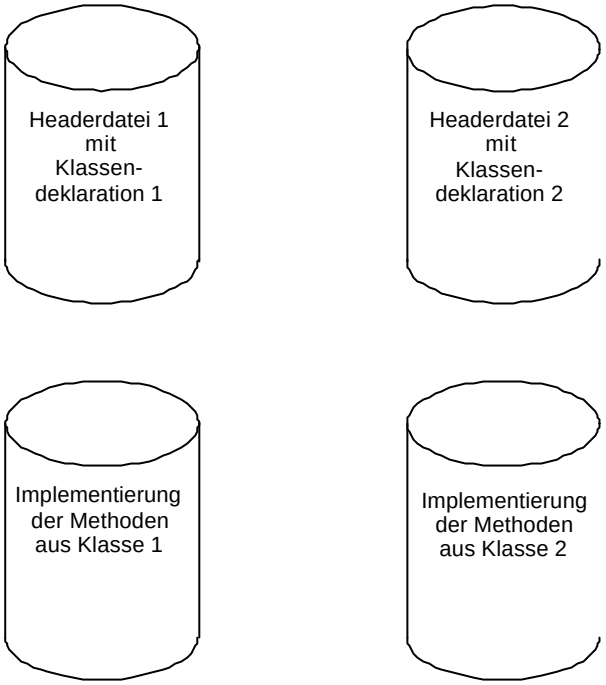


Bild: Zusammenspiel von Header- Implementierungs- und Hauptprogrammdatei.

Konstruktoren und Destruktoren

Die runden Klammern am Ende der einzelnen Zeilen, deuten auf die Übergabe von Parametern an eine Funktion hin und an dieser Stelle wird tatsächlich eine spezielle Funktion aufgerufen. Jedesmal, wenn in einem C++ Programm ein Objekt einer Klasse erzeugt wird, dann wird genau zu diesem Zeitpunkt eine spezielle Funktion, der sogenannte "Konstruktor" aufgerufen. Diese Funktion findet sich auch in der Klassendeklaration wieder und trägt immer exakt den gleichen Namen, wie die Klasse. Ihr Gegenstück ist der "Destruktor", der genau dann aufgerufen wird, wenn ein Objekt gelöscht wird. In unserem einfachen Beispiel ist er leer, er bekommt aber größere Bedeutung wenn ein Objekt dynamisch erzeugt werden soll, es also über "Pointer" angesprochen wird, oder es "Pointer" auf andere Objekte enthält. In diesen Fällen muß der belegte Speicher "von Hand", beim Löschen des Objekts wieder freigegeben werden. Der Zusammenhang von Konstruktoren und Destruktoren mit der Erzeugung und Vernichtung von Objekten ist nächsten Bild dargestellt:

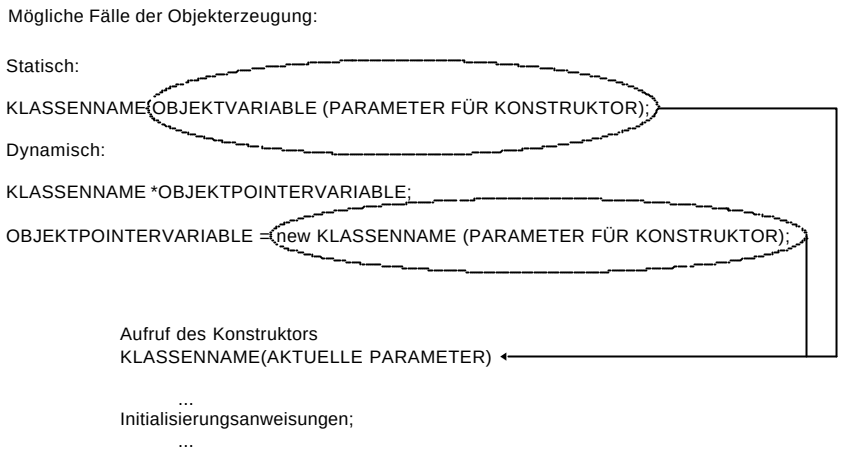


Bild: Zusammenhang von Konstruktoren und Destruktoren mit der Erzeugung und Vernichtung von Objekten.

Im Gegensatz zum Destruktor ist der Konstruktor in unserem Beispiel auch implementiert. Selbst wenn man keine der beiden Funktionen bei der Deklaration angibt, so werden sie trotzdem aufgerufen und ausgeführt, auch wenn der Programmierer das dann nicht erkennen kann. Ein Konstruktor erfüllt den Zweck, beim Erzeugen eines Objekts bestimmte "Member-Variablen" auf einen definierten Startwert zu setzen. Häufig wird z.B. für "int"-Variablen der Wert 0 angesetzt oder aber es wird ein Startwert explizit übergeben, was im Konto-Beispiel der Fall ist. Unser Konstruktor ist so deklariert, daß er zwei Parameter erhält, nämlich die Kontonummer (vom Typ `long`) und den Betrag (vom Typ `double`), den das Konto bei seiner Eröffnung, also der Erzeugung des Kontoobjektes enthalten soll. Die konkreten Werte für die Parameter werden also im Hauptprogramm direkt an den Konstruktor übergeben, der sie dann den beiden "Member-Variablen" zuweist. Dieses Vorgehen ist typisch für C++ und jede gut entworfene Klasse sollte einen Konstruktor enthalten, um eine vernünftige Initialiserung durchzuführen. Man sollte sich nie darauf verlassen, daß z.B. ein int-Member vom Compiler beim Anlegen mit 0 initialisiert wird. Das kann zwar der Fall sein, in der Regel stehen aber völlig zufällige Werte in den "Member-Variablen". Es ist außerdem durchaus möglich und üblich mehrere Konstruktoren zu implementieren. Sind mehere Konstruktoren vorhanden, so müssen sie sich durch ihre Parameterliste eindeutig unterscheiden lassen. Ein Konstruktor mit leerer Parameterliste wird als "Default-Konstruktor" bezeichnet. Er wird standardmäßig aufgerufen, wenn ein Objekt ohne die Angabe von Parametern instanziiert wird. Beim Erzeugen eines Objekts entscheidet der Compiler dann anhand der aktuellen Parameterliste welcher Konstruktor aufgerufen wird. Dieses Verfahren entspricht dem in Kapitel 0 beschriebenen "Overloading".

Die "."-Notation

In der Zeile "k1.kontostand_ausgeben();" im Hauptprogramm wird über die "." Notation auf ein Element eines Objektes zugegriffen. In diesem Fall wird die "Member-Funktion" "kontostand_ausgeben()" aufgerufen. Durch die Notation "Objekt.Element" kann man auf einzelne Elemente eines Objektes zugreifen, unter der Bedingung, daß diese Elemente zugreifbar sind, d.h. daß sie "public" deklariert sind. Würde man etwa versuchen im Hauptprogramm auf die Member-Variable "m_dBetrag" zuzugreifen, so würde der Compiler beim Übersetzen eine Fehlermeldung ausgeben. Diese Variable ist "private" deklariert und deshalb nicht von außen sichtbar. Näheres über diese Schutzfunktionen und die dahinterstehenden Konzepte findet sich im nächsten Kapitel.

Dynamische Objekte

In Kapitel 0 wurde bereits auf die dynamische Speicherverwaltung eingegangen. Da Klassen im Prinzip nichts anderes als benutzerdefinierte Datentypen darstellen, kann man Variablen eines Klassentyps, also Objekte, genauso dynamisch erzeugen wie Variablen eines anderen Typs. Als Beispiel werden wieder drei Kontoobjekte erzeugt, allerdings sollen sie diesmal nicht mehr in einzelnen Objekt-Variablen abgelegt werden sondern es soll ein Array (s. Kapitel 0 und 0 aus dem C-Skript) mit drei Einträgen

angelegt werden, wobei jeder Array-Eintrag ein Pointer auf ein Kontoobjekt darstellt (s. Bild kontopointarray.wmf). Der Quellcode für ein solches Vorgehen sieht dann so aus:

```
// Hauptprogramm zur Kontoverwaltung in einem Array
aus Pointern
#include "iostream.h"
#include "stdlib.h"
#include "konto1.h" // Einfügen der Header-Datei in
bank.cpp
main()
{
 int i;
 // Anlegen eines Arrays aus Pointern auf
Kontoobjekte
 Konto *kontoArray[3];
 // Erzeugen von 3 Objekten der Klasse Konto
 for(i=0;i<=2;i++)
    kontoArray[i] = new Konto(i,100.0);
 // Ausgeben der aktuellen Kontostände der 3 Objekte
 kontoArray[0]->kontostand_ausgeben();
 kontoArray[1]->kontostand_ausgeben();
 kontoArray[2]->kontostand_ausgeben();
 // 50.- von kontoArray[0] abheben und dann den
neuen Stand ausgeben
 kontoArray[0]->abheben(50.0);
 kontoArray[0]->kontostand_ausgeben();
 // Versuch 100.- von k2 abzuheben (geht nicht, kein
Kredit)
 kontoArray[1]->abheben(100.0);
 // 75.- auf k2 einzahlen und Kontostand ausgeben
 kontoArray[1]->einzahlen(75.0);
 kontoArray[1]->kontostand_ausgeben();
 // Erneuter Versuch 100.- abzuheben (diesmal klappt
es)
 kontoArray[1]->abheben(100.0);
```

```
    kontoArray[1]->kontostand_ausgeben();
    // Manipulation und Ausgabe des Standes von k3
    kontoArray[2]->einzahlen(10000.0);
    kontoArray[2]->abheben(5000.0);
    kontoArray[2]->kontostand_ausgeben();
    // Ordnungsgemäßes Löschen des Kontoarrays
    for(i=0;i<=2;i++)
       delete kontoArray[i];
    cin >> i;
    exit(0); // Programm ordnungsgemäß verlassen
   }
```

Man erkennt, daß zunächst ein Array mit drei Elementen (0..2, vgl. Kapitel 0) von Pointern auf Kontoobjekte angelegt wird. Diese Pointer zeigen zunächst noch ins Leere, die eigentliche Erzeugung der konkreten Objekte findet erst in den drei folgenden Zeilen statt. Hier kommt ein weiteres C++ Schlüsselwort ins Spiel, der Operator `new`. Er dient als Ersatz für die in C benutzten `malloc`- und `calloc`-Funktionen. Er alloziert Speicher für ein Objekt des gewünschten Typs, wobei der entsprechend Typ hinter dem Operator angegeben wird. In unserem Fall ist der Typ die Klasse "Konto". In der "for"-Schleife wird jedem Array-Element (also den Pointern) das frisch erzeugte Objekt zugewiesen. Anschließend zeigt jedes Arrayelement auf ein Objekt der Klasse Konto (s. a. Bild Zugriff auf Objekte über Array aus "Pointern"). Auch hier wird wieder der Konstruktor aufgerufen, jetzt mit den Parametern "i" (die Kontonummern sind einfach die fortlaufenden Array-Indizes) und dem festen Betrag von 100.-.

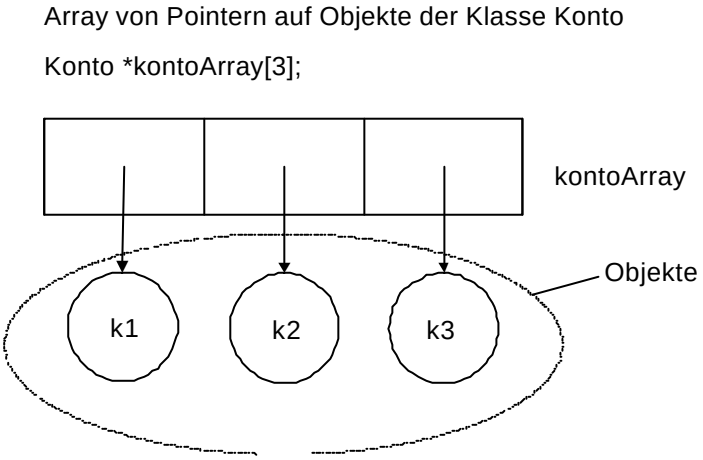


Bild: Ein Array aus "Pointern" auf Objekte.

Nach der Initialisierung des Objekt-Arrays werden im Prinzip die gleichen Operationen und Funtkionsaufrufe getätigt, wie im Beispiel zuvor. Natürlich wird jetzt über das Pointer-Array auf die Objekte und ihre Member-Funktionen zugegriffen. Dabei muß man eine andere Notation verwenden. Statt des Punktes wird nun die Pfeilnotation ("->") verwendet. Dies ist immer dann notwendig, wenn auf ein Element eines Objektes über einen Pointer und nicht direkt zugegriffen wird. Abgesehen von diesem Unterschied ist das Vorgehen und das Ergebnis des Hauptprogrammes exakt das gleiche, wie im ersten Beispiel. Die Erzeugung und der Zugriff auf Objekte über Pointer in C++ ist eine übliche Vorgehensweise, da dieses Verfahren die Ausführungsgeschwindigkeit von Programmen erhöht und den

Speicherbedarf reduziert. Bei der Übergabe von Pointern auf Objekte an Funktionen werden z.B. nur die Pointer übergeben, nicht aber die ganzen Objekte.

Zusammenfassung

In diesem Kapitel wurden die Begriffe Klasse und Objekt erläutert, voneinander abgegrenzt und ihre programmtechnische Umsetzung in C++ beschrieben. Es mag vielleicht sehr aufwendig aussehen, Klassen und Objekte bei der Programmierung zu verwenden, hätte man doch das Konto-Beispiel auch mit einfachem C schnell und elegant realisieren können. Es sei an dieser Stelle darauf verwiesen, daß die objektorientierte Programmierung allgemein und C++ im Speziellen, explizit für die Entwicklung von sehr großen Softwareprodukten entworfen worden sind. In solchen Projekten bietet die Objektorientierung eine optimale Unterstützung. Der Analyse- und Designprozeß, d.h. die Zerlegung des Problemfeldes in Klassen und die Bestimmung der Interaktion der Objekte untereinander stellt dabei die eigentliche Problematik dar. Sind diese Schritte sauber durchgeführt ist die eigentliche Programmierung relativ simpel.

Member-Funktionen, Zugriffsrechte und andere Konstrukte

Im letzten Kapitel sind die Grundkonzepte der objektorientierten Programmierung (Klassen und Objekte) bereits eingeführt worden. Dieses Kapitel wird sich noch einmal näher mit der Bedeutung von Methoden ("Member-Funktionen") und der verschiedenen Zugriffsbeschränkungen, insbesondere im Hinblick auf das Schlagwort "Information Hiding", beschäftigen. Anhand eines einfachen Beispiels wird die Möglichkeit erläutert, Objekte verschiedener Klassen zu benutzen, ohne das man den Aufbau der Datenstrukturen innerhalb eines Objekts kennen muß. Zusätzlich wird das Konzept der "friends" vorgestellt, mit dem einige Zugriffseneinschränkungen gezielt wieder aufgehoben werden können. Außerdem werden die beiden C++ Konstrukte "Referenzen" und "Konstanten" vorgestellt, die es in dieser Form in reinem C nicht gibt.

Member-Funktionen zum Zugriff auf Attribute

Der normale Weg in C++ um von "außen" auf ein Attribut eines Objekts zuzugreifen ist der, daß man eine oder mehrere Methoden definiert, die den Wert des Attributs lesen bzw. schreiben können. Das folgende Beispiel zeigt dieses Vorgehen anhand des Beispiels einer Klasse für die rechnerinterne Darstellung von Rechtecken. Dabei werden nacheinander zwei verschiedene Klassen entworfen, die jeweils Rechteckobjekte beschreiben. Die interne Darstellung der Koordinaten ist dabei unterschiedlich, der Benutzer, (hier die `main()`-Funktion) "merkt" gar nicht, daß sich die interne Darstellung geändert hat. Man kann die Klassen einfach austauschen, denn solange sich die Zugriffsfunktionen, d.h. das "Interface" der Klasse nicht verändern, ist es nicht nötig das Hauptprogramm zu anzupassen. Hier nun die Klassendeklaration und -Implementierung, sowie das zugehörige Hauptprogramm zum Test.

Die Deklaration der Klasse "Rec":

```
#ifndef _REC_H_
#define _REC_H_

class Rec
{
 // Methoden
 public:
 // Konstruktion
        Rec();
        Rec(const double &,const double &,const
double &,const double &);
   // Ausgabe auf Bildschirm
   void printCoordinates();
   void printPlaneContent();
   void setCoordinates(const double &,const double
&,const double &,const double &);
 private:
```

```
   // 1. Implementierung der Attribute
   // nur untere linke und obere rechte Ecke werden
gespeichert
   double m_dLowerLeftX,m_dLowerLeftY;
   double m_dUpperRightX,m_dUpperRightY;
};
#endif

Die Implementierung der Klasse "Rec"

// Implementierung der Methoden von Rec
#include "iostream.h"
#include "rec.h"

// default Konstruktor
Rec::Rec()
{
 m_dLowerLeftX = 0.0;
 m_dLowerLeftY = 0.0;
 m_dUpperRightX = 0.0;
 m_dUpperRightY = 0.0;
}

// Konstruktor mit dem explizit Koordinaten gesetzt
werden koennen
Rec::Rec(const double &llx,const double &lly,const
double &urx,const double &ury)
{
 m_dLowerLeftX = llx;
 m_dLowerLeftY = lly;
 m_dUpperRightX = urx;
 m_dUpperRightY = ury;
}
```

```
// Ausgabe des Flaecheninhalts auf dem Bildschirm
void Rec::printPlaneContent()
{
 double pc;

 pc = (m_dUpperRightX - m_dLowerLeftX) *
(m_dUpperRightY - m_dLowerLeftY);
 cout << "Flaecheninhalt des Rechtecks: " << pc <<
"\n";
 return;
}

// Ausgabe der Koordinaten des Rechtecks auf dem
Bildschirm
void Rec::printCoordinates()
{
 cout << "Untere linke Ecke X: " << m_dLowerLeftX <<
" Untere linke Ecke Y: " << m_dLowerLeftY << "\n";
 cout << "Untere rechte Ecke X: " << m_dUpperRightX
<< " Untere rechte Ecke Y: " << m_dLowerLeftY <<
"\n";
 cout << "Obere rechte Ecke X: " << m_dUpperRightX
<< " Obere rechte Ecke Y: " << m_dUpperRightY <<
"\n";
 cout << "Obere linke Ecke X: " << m_dLowerLeftX <<
" Obere linke Ecke Y: " << m_dUpperRightY << "\n";
}

// Setzen der Rechteckkoordinaten durch Angabe von
unterer linker und oberer rechter Ecke
void Rec::setCoordinates(const double &llx,const
double &lly, const double &urx, const double &ury)
```

```
{
 m_dLowerLeftX = llx;
 m_dLowerLeftY = lly;
 m_dUpperRightX = urx;
 m_dUpperRightY = ury;
}
```

Das Hauptprogramm:

```
#include "iostream.h"
#include "conio.h"
#include "rec.h"

int main()
{
 Rec rec1; // Konstruktion ueber Default Konstruktor
 Rec rec2(1.0,1.0,10.0,10.0); // Konstruktion ueber
anderen Konstruktor
 // Bildschirmausgabe
 clrscr();
 rec1.printCoordinates();
 rec1.printPlaneContent();
 rec2.printCoordinates();
 rec2.printPlaneContent();
 // Umsetzen der Koordinaten von rec2
 rec2.setCoordinates(1.0,1.0,5.0,5.0);
 rec2.printCoordinates();
 rec2.printPlaneContent();
 return 0;
}
```

Im wesentlichen werden in der Klasse "Rec" nur bereits bekannte Standard-Konstrukte verwendet, bis auf den Referenzoperator "&", auf den weiter unten eingegangen wird. Wichtig ist an dieser Stelle, daß in dieser Implementierung der Klasse "Rec" nur die linke unter und die rechte obere Ecke des Rechtecks gespeichert werden. Soll der Flächeninhalt ausgegeben werden, müssen die Kantenlängen jeweils extrahiert werden, genau wie bei der Ausgabe der Koordinaten, bei der die beiden fehlenden Eckpunkte

aus den gespeicherten generiert werden müssen. Die Member-Variablen sind als `private` deklariert, was generell immer so gehalten werden sollte, um sie vor Zugriffen von "außen" zu schützen. Es existiert außer `public` und `private` noch eine weitere Sicherheitsstufe, nämlich `protected` in C++, die in Zusammenhang mit der Technik der Vererbung (s. Kapitel 12) von Bedeutung ist.

Alternative Implementierung der Klasse "Rec"

Man stelle sich jetzt vor, der Entwickler der Klasse "Rec" beschließt die Implementierung dahingehend zu ändern, alle vier Eckpunkte direkt abzuspeichern. Dieses Vorgehen benötigt zwar etwas mehr Speicher, aber dafür ist der Zugriff auf die Daten zur Ausgabe und zur Flächeninhaltsberechnung schneller. Würde man die Member-Variablen öffentlich zugänglich machen und im Hauptprogramm, ohne den "Umweg" über Methoden der Klasse auf diese Variablen zugreifen, dann müßte die Implementierung des Hauptprogramms geändert werden. Außerdem müßte der Entwickler des Hauptprogramms (oder die Entwickler anderer Programmteile oder Klassen, die Objekte der Klasse "Rec" benutzen wollen) den internen Aufbau der Klasse "Rec" kennen. Bei Verwendung von Member-Funktionen als öffentliche Schnittstelle ist diese interne Kenntnis nicht notwendig. Bei der Entwicklung von großen Programmen ist eine solche interne Kenntnis von anderen Programmteilen nicht erwünscht und man versucht bewußt, nur über genau definierte Schnittstellen andere Objekte zu benutzen. Dieses Verstecken von Informationen bezeichnet man in der Informatik als "Information Hiding" bezeichnet und es wird durch objektorientierte Programmierung gut unterstützt, wie man auch im nächsten Bild sieht.

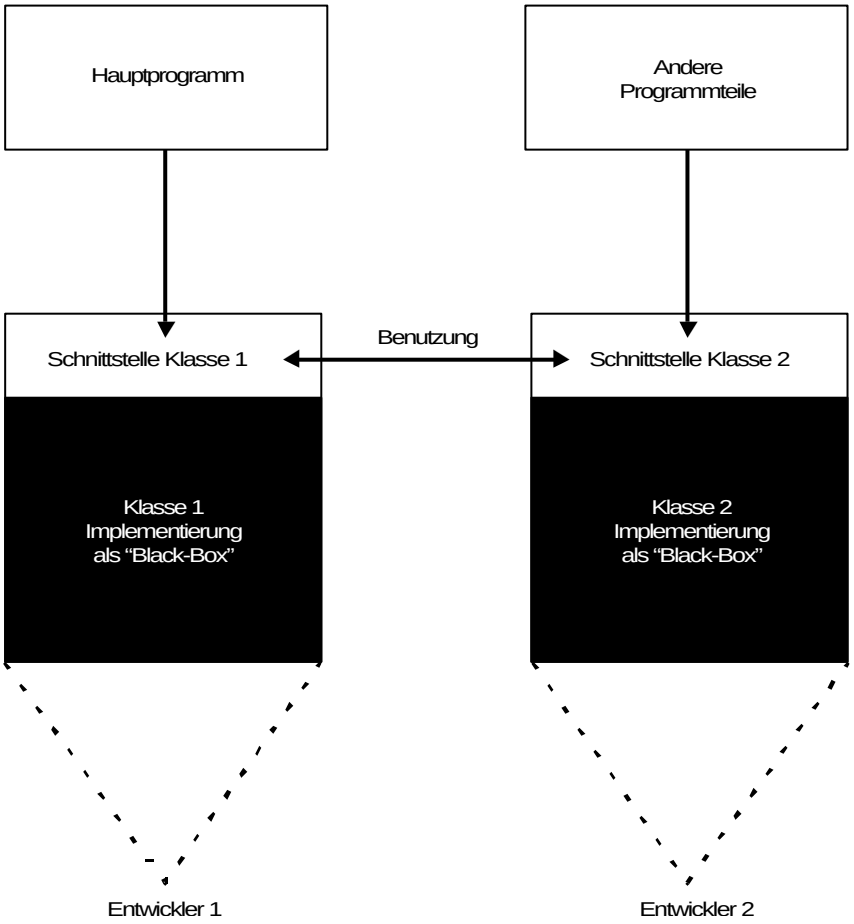


Bild: Konzept des Information-Hiding in der Programmierung mit C++

Während in unserem einfachen Beispiel der Umweg des Zugriffs über Methoden eher umständlich erscheint, ist dieses Vorgehen bei größeren Programmen sehr sinnvoll. Oftmals ist es z.B. so, daß der Hersteller einer IDE eine sogenannte Klassenbibliothek mitliefert. Von einer solchen Bibliothek bekommt

der Anwender lediglich die Headerfiles mit den Zugriffsfunktionen zu sehen, während die Implementierungsfiles der einzelnen Klassen bereits binär übersetzt geliefert werden. Dadurch ist es nicht notwendig, daß der Anwender die Interna der Bibliothek kennt, sie aber trotzdem benutzen kann. Ein bekanntes Beispiel für eine solche Klassenbibliothek sind die "Microsoft Foundation Classes", kurz "MFC", die dem Programmierer eine umfangreiche und benutzerfreundliche Klassensammlung zur Programmierung von "Windows®"-Applikationen zur Verfügung stellen.
Bei unserem Programm wird jetzt der Quelltext der neuen Implementierung der Klasse "Rec" angegeben. Man beachte, daß nur das "Rec.cpp"-File und das "Rec.h"-File ausgetauscht werden müssen, das Hauptrogramm bleibt exakt so, wie es ist. Die Implementierung ändert sich, aber die Schnittstelle bleibt die gleiche. Hier nun der modifizierte Quelltext, zunächst das Headerfile mit der Klassendeklaration:

```
#ifndef _REC_H_
#define _REC_H_

class Rec
{
 // Methoden, hier aendert sich nichts
 public:
 // Konstruktion
        Rec();
        Rec(const double &,const double &,const
double &,const double &);
   // Ausgabe auf Bildschirm
   void printCoordinates();
   void printPlaneContent();
   void setCoordinates(const double &,const double
&,const double &,const double &);
 private:
   // 2.Implementierung der Attribute
   // Untere linke Ecke wird gespeichert (wie
bisher)
   double m_dLowerLeftX,m_dLowerLeftY;
   // Neu: jetzt wird auch die Obere linke Ecke
gespeichert
   double m_dUpperLeftX,m_dUpperLeftY;
   // Neu: jetzt wird auch die untere rechte Ecke
gespeichert
   double m_dLowerRightX, m_dLowerRightY;
```

```
   // Obere rechte Ecke wird gespeichert (wie
bisher)
   double m_dUpperRightX,m_dUpperRightY;
};
#endif

Im folgenden die modifizierte Implementierung der
Klasse "Rec":

// Implementierung der Methoden von Rec
#include "iostream.h"
#include "rec1.h"

// default Konstruktor
Rec::Rec()
{
 m_dLowerLeftX = 0.0;
 m_dLowerLeftY = 0.0;
 m_dUpperLeftX = 0.0; // Neu: x-Koordinate der
oberen linken Ecke wird initialisiert
 m_dUpperLeftY = 0.0; // Neu: y-Koordinate der
oberen linken Ecke wird inittalisiert
 m_dLowerRightX = 0.0; // Neu: x-Koordinate der
unteren rechten Ecke wird initialisiert
 m_dLowerRightY = 0.0; // Neu: y-Koordinate der
unteren rechten Ecke wird initialisiert
 m_dUpperRightX = 0.0;
 m_dUpperRightY = 0.0;
}

// Konstruktor mit dem explizit Koordinaten gesetzt
werden koennen
Rec::Rec(const double &llx,const double &lly,const
```

```
double &urx,const double &ury)
{
 m_dLowerLeftX = llx;
 m_dLowerLeftY = lly;
 m_dUpperRightX = urx;
 m_dUpperRightY = ury;
 // Neu Initialisierung der neuen Eckpunkte
 m_dUpperLeftX = llx;
 m_dUpperLeftY = ury;
 m_dLowerRightX = urx;
 m_dLowerRightY = lly;
}

// Ausgabe des Flaecheninhalts auf dem Bildschirm
void Rec::printPlaneContent()
{
 double e1,e2,pc;
 // Neu: Berechnung des Flaecheninhalts durch
Berechnung der Kantenlaengen
 // und anschliessender Multiplikation

 e1 = m_dLowerRightX - m_dLowerLeftX;
 e2 = m_dUpperLeftY - m_dLowerLeftY;
 pc = e1*e2;
 cout << "Flaecheninhalt des Rechtecks: " << pc <<
"\n";
 return;
}

// Ausgabe der Koordinaten des Rechtecks auf dem
Bildschirm
void Rec::printCoordinates()
{
```

```
 // Neu: Fuer jede Ausgabe wird jetzt eigene Member-
Variable benutzt
 cout << "Untere linke Ecke X: " << m_dLowerLeftX <<
" Untere linke Ecke Y: " << m_dLowerLeftY << "\n";
 cout << "Untere rechte Ecke X: " << m_dLowerRightX
<< " Untere rechte Ecke Y: " << m_dLowerRightY <<
"\n";
 cout << "Obere rechte Ecke X: " << m_dUpperRightX
<< " Obere rechte Ecke Y: " << m_dUpperRightY <<
"\n";
 cout << "Obere linke Ecke X: " << m_dUpperLeftX <<
" Obere linke Ecke Y: " << m_dUpperLeftY << "\n";
}

// Setzen der Rechteckkoordinaten durch Angabe von
linker unterer und rechter oberer Ecke
void Rec::setCoordinates(const double &llx,const
double &lly, const double &urx, const double &ury)
{
 m_dLowerLeftX = llx;
 m_dLowerLeftY = lly;
 m_dUpperRightX = urx;
 m_dUpperRightY = ury;
 // Neu: Setzen der neuen Koordinaten
 m_dUpperLeftX = llx;
 m_dUpperLeftY = ury;
 m_dLowerRightX = urx;
 m_dLowerRightY = lly;
}
```

Wenn man die beiden Dateien gegen die alten Versionen im Projekt austauscht, dann ändert sich am Verhalten des Hauptprogramms nichts. Lediglich der interne Aufbau der Klasse "Rec" hat sich geändert, was sich aber in der Schnittstelle nicht niederschlägt. Deshalb können Objekte dieser Klasse genauso verwendet werden, wie vorher.

Friends

Nachdem wir gerade die Technik zum Verstecken von Informationen kennengelernt haben, folgt jetzt eine Methode, mit der man diese Sicherheitsmechanismen gezielt außer Kraft setzen kann. Man sollte das Konstrukt der "friend"-Klassen nur sehr sparsam verwenden und wenn es sich umgehen läßt am besten überhaupt nicht. Allerdings kann es manchmal vorkommen, daß vor allem aus Geschwindigkeitsgründen, Zugriffe auf interne Datenstrukturen nicht über Methoden erfolgen können, sondern direkt durchgeführt werden müssen. Um dieses Ziel zu erreichen kann eine Klasse eine andere als `friend` deklarieren. Die so deklarierte zweite Klasse hat dann (ausnahmsweise) auch Zugriff auf die privaten Datenstrukturen der ersten Klasse. Ein Anwendungsfall sind z.B. Status-Objekte, wie sie in der Computergrafik verwendet werden, und die die gesamten Grunddaten einer "Grafikpipeline" enthalten. Auf diese Daten wird in der Regel sehr oft zugegriffen und die Zugriffe müssen gerade in der Computergrafik möglichst schnell ablaufen. An dieser Stelle ist der Zugriff über eine Methode zu langsam, da dabei jedesmal ein Funktionsaufruf durchgeführt werden muß, wobei dann womöglich auch noch Parameter übergeben werden müssen. Hier nun ein einfaches Beispiel zur Deklaration von "friends":

```
// Headerdatei zur Klasse A
#ifndef _A_H_
#define _A_H_

#include "iostream.h"
#include "b.h"

class A
{
  // Konstruktor
  public:
  A(const int&);
  // Methode zur Summenbildung
  void SumAPlusB(const B&);

  // Privates Attribut zu Demo-Zwecken
  private:
  int m_iWert;
};
#endif

// Implementierung von A
#include "a.h"
#include "b.h"
```

```
// Konstruktor
A::A(const int &init)
{
 m_iWert = init;
}

// Druckt die Summe des Werte eines Objektes von A
und B auf den Bildschirm
// Direkter Zugriff auf private-Element von b, weil
B als friend deklariert ist
void A::SumAPlusB(const B &b)
{
 cout << "Summe von A und B: " << m_iWert +
b.m_iWert << "\n";
}

// Headerdatei zur Klasse B
#ifndef _B_H_
#define _B_H_

class B
{
  // Die Klasse B hat die Klasse A zum friend
  // und erlaubt Objekten der Klasse A den Zugriff
auf private Datenstrukturen
  friend class A;
  // Konstruktor
  public:
  B(const int&);

  // Privates Attribut zu Demo-Zwecken
  private:
```

```
  int m_iWert;
};
#endif

// Implementierung von B
#include "b.h"

// Konstruktor
B::B(const int &init)
{
 m_iWert = init;
}

// Hauptprogramm
#include "iostream.h"
#include "a.h"
#include "b.h"

int main()
{
 // Erzeugen zweier Objekte a und b
 A a(10);
 B b(10);

 // Aufruf der Summenmethode von A
 a.SumAPlusB(b);
 exit(0);
}
```

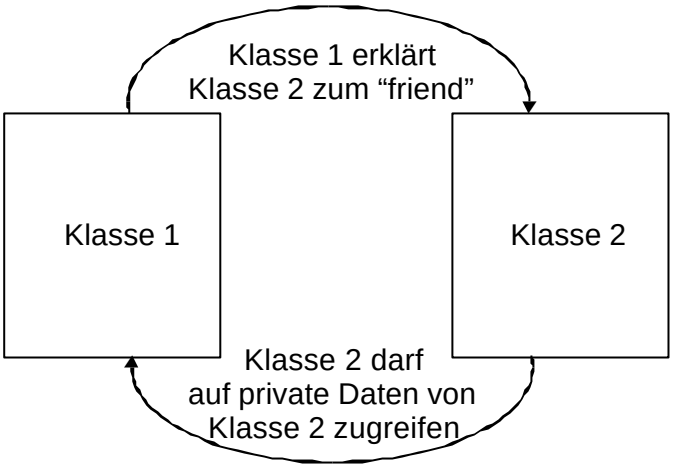
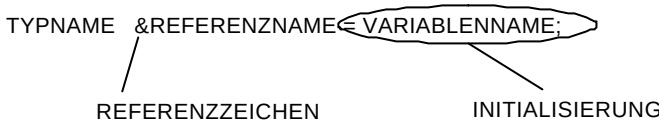


Bild: Konzept der friend-Klassen.

In obigem Beispiel deklariert die Klasse "B", die Klasse "A" als "friend". D.h., daß A auf "private"-Elemente von B zugreifen darf und damit den Schutz von privaten Datenstrukturen (die "Kapselung") umgehen darf. Das wird in der Methode "SumAPlusB" ausgenutzt, die als Parameter ein Objekt der Klasse "B" erwartet. In der Methode wird dann direkt auf die Member-Variable "m_iWert" des Objekts zugegriffen obwohl sie "private" deklariert wurde. Allgemein ist zu beachten, daß eine Klasse nur von sich aus andere Klassen zu "friends" erklären und damit den Zugriff auf ihre internen Strukturen erlauben kann. Der umgekehrte Weg, nämlich, daß eine Klasse sich als "friend" einer anderen erklärt ist nicht möglich. Es ist außerdem möglich, auch einzelne Methoden zu "friends" zu erklären, an dieser Stelle sei aber auf weiterführende Literatur verwiesen, wie etwa [10].

Referenzen

In den gerade vorgestellten Beispielen taucht immer wieder das Zeichen "&" auf, das man als "Referenzoperator" bezeichnet. In reinem C besitzt dieses Konstrukt eine andere Bedeutung (Adressoperator), während es in C++ dazu benutzt wird, um Kopien von Variablennamen zu erzeugen. Wie bereits in Kapitel 0 des C-Kurses erläutert wurde, wird bei der Übergabe einer Variable an eine Funktion immer nur eine Kopie innerhalb der Funktion angelegt. Verändert man also diese Kopie, so ändert sich der Variablenwert außerhalb der Funktion überhaupt nicht. Möchte man einen Parameterwert innerhalb einer Funktion oder Methode manipulieren, so mußte man bislang einen "Pointer" auf die entsprechende Variable übergeben. In C++ ist es aber auch möglich, eine Referenz auf ein Objekt anzulegen und dann diese Referenz zu übergeben. Eine Referenz stellt dabei nichts weiter dar, als einen neuen Namen für ein bereits deklariertes Objekt. Das folgende Bild und der anschließende Sourcecode zeigen einige Beispiele für die Benutzung von Referenzen:



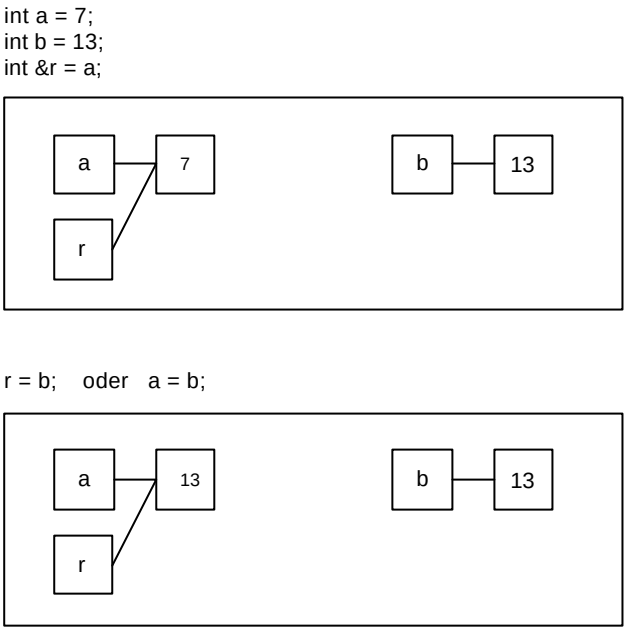


Bild: Definition und Benutzung von Referenzen

```
// Hauptprogramm zur Definition von Referenzen
#include "iostream.h"
#include "conio.h"

// Funktionsdeklaration
void testFunc1(int);
void testFunc2(int &);

int main()
{
 int a,c;    // Normale Integer-Variable
 int &b = a; // b als Referenz (zweiter Name) auf a

 // Bildschirm loeschen
 clrscr();
```

```
 // Initialisierung
 a = 5;    // a bekommt den Wert 5
 b = 10;    // b bekommt neuen Wert, gleichzeitig
auch a
 c = 15;
 cout << "Wert von a: " << a << " Wert von b: " << b
<< "\n";

 // Aufruf bei normaler Variablenuebergabe, alter
Wert von c im Hautprogramm
 cout << "Normale Variablenuebergabe \n";
 cout << "Wert von c vor dem Funktionsaufruf: " << c
<< "\n";
 testFunc1(c);
 cout << "Wert von c nach dem Funktionsaufruf: " <<
c << "\n";

 // Aufruf mit Referenz, neuer Wert auch im
Hauptprogramm
 cout << "Uebergabe per Referenz: \n";
 cout << "Wert von c vor dem Funktionsaufruf: " << c
<< "\n";
 testFunc2(c);
 cout << "Wert von c nach dem Funktionsaufruf: " <<
c << "\n";
 exit(0);
}

// Beispielfunktion zur standardmaessigen Uebergabe
von Variablen
void testFunc1(int in)
{
```

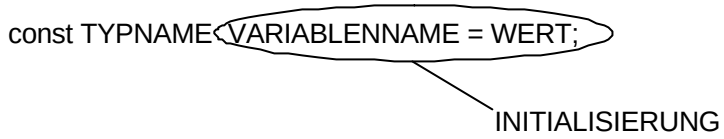
```
 int a = 5;
 in = a;  // Veraenderung des uebergebenen
Parameters
 return;
}

// Beispielfunktion zur Manipulation einer Referenz
in einer Funktion
void testFunc2(int &inRef)
{
 int a = 5;
 inRef = a; // Veraenderung des Referenzwertes
 return;
}
```

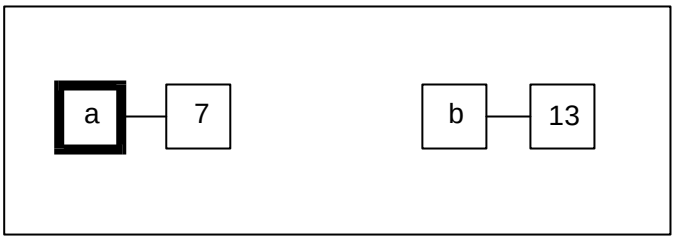
Man beachte, daß man die Begriffe "Referenz" und "Pointer" nicht synonym behandeln darf. Eine "Referenz" ist tatsächlich ein zusätzlicher Variablenname für ein bereits deklariertes Objekt, während ein "Pointer" auf die Adresse eines Objekts zeigt. Besondere Bedeutung kommt Referenzen in Zusammenhang mit überladenen Operatoren zu, wie sie in Kapitel 0 beschrieben werden. Auch an dieser Stelle sei wieder auf weiterführende Literatur verwiesen, z.B. [10].

Konstanten

Als letzter Teil in diesem Kapitel wird im folgenden das bereits verwendete C++-Sprachkonstrukt "const", erläutert. "const" erlaubt die Deklaration (und Definition) von konstanten Objekten, also von Objekten denen außer bei ihrer Initialisierung, im weiteren Programmverlauf keine neuen Werte zugewiesen werden können. Ein solches Objekt bleibt im Verlauf seiner Lebensdauer konstant. In C wurde dieses Verhalten durch die Verwendung des Präprozessor-Statements `#define` erreicht. Da dieses Statement jedoch eine reine Textersetzung durchführt, bietet es keine Möglichkeit für Typprüfungen. Außerdem kann man sich Werte, die mit `#define` definiert wurden nicht im Debugger ansehen, was die Fehlersuche oft erschwert. Das nächste Beispielprogramm und das nächste Bild zeigen die Deklaration und einige Anwendungen für konstante Variablen. Selbstverständlich lassen sich auch selbstdefinierte Objekte als konstant deklarieren, wie das Beispiel der Objektvariable "d" zeigt.




```
const int a = 7;
int b = 13;
```




```
a = b; // Fehler, einer Konstanten darf man nichts zuweisen
```

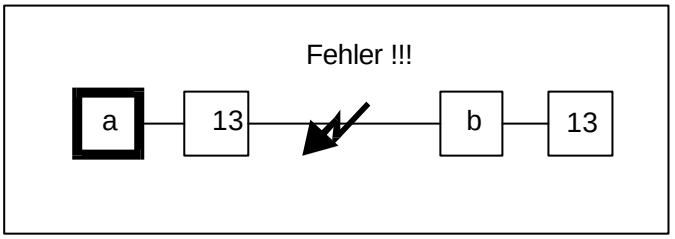


Bild: Verhalten von Variablen, die als konstant deklariert worden sind.

```
// Hauptprogramm zur Definition von Referenzen
#include "iostream.h"
#include "conio.h"

// Ausnmahmsweise Klassendeklaration in cpp-File
class ConstTest
{
 // Attribute
 public:
         int m_iA;
 // Methoden
 public:
        ConstTest(int in){m_iA=in;}
};
```

```
int main()
{
 const int a = 10;    // Konstante Integer-Variable
mit Init
 const double b = 3.141592; // Konstante double
Variable mit Init
 // const double c; // Konstante double Variable
ohne Init: Fehler!!!
 const double c = 6.5; // Konstante double Variable
mit Init: Kein Fehler!!!
 const ConstTest d(10); // Konstantes Objekt der
Klasse ConstTest
 // Bildschirm loeschen
 clrscr();

 // Initialisierung
 // a = 5;    // Versuch einer Zuweisung:
Fehlermeldung !!!
 cout << "Wert von a: " << a << " Wert von b: " << b
<< " Wert von c: " << c << "\n";
 cout << "Member-Variable a des Objekts d: " <<
d.m_iA << "\n";
 // d.m_iA = 5 // Versuch des schreibenden Zugriffs
auf m_iA Fehler !!
 exit(0);
}
```

Nach den ersten beiden Kapiteln des C++-Kurses sind die wesentlichen Grundlagen zur objektorientierten Programmierung in C++ gelegt. Die nächsten beiden Kapitel stellen weitergehende Konzepte vor und vertiefen das bisher gelernte.

Überladen von Operatoren und Funktionen

Überladen von Funktionen

Durch das Überladen von Funktionen können unterschiedliche Funktionen mit dem gleichen Funktionsbezeichner, aber verschiednen Argumenten aufgerufen werden.
Der Vorteil dieser Vorgehensweise liegt in der Vereinfachung der eigentlichen Implementation, da der verwendete Datentyp für eine Funktionalität häufig irrelevant ist. Das Verhalten der Funktion bleibt identisch, egal auf welche Datentypen diese angewendet wird.

Beispiel:

Die Funktion "+" kann in der Mathematik auf natürliche, reelle und komplexe Zahlen oder Vektoren angewendet werden. Die Funktionalität bleibt gleich (die Addition), während die Funktion auf unterschiedliche Datentypen (natürlich, reell, komplex, Vektor) auch gemischt angewendet werden kann. In C++ ist der Operator "+" für `int` (ganze Zahlen) und `float` bzw. `double` (reelle Zahlen) überladen, d.h. für die unterschiedlichen "+"-Operationen existiert nur ein Symbol.

Beispiel:

```
int    maximum  (int, int); // Prototyp der Funktion
//maximum zweier int-Zahlen
int    maximum  (const int*,  int); // hier für ein
//Array und eine int-Zahl
int         j = 10;
int         k = 12;
int              a[4] =  {5,7,9,1};
  .
  .
int    main()
{
       cout << "Maximum von Array und Einzelzahl" <<
maximum (a, k)  << endl;
       cout << "Maximum Int und Int " <<  maximum (j
, k)  <<  endl;
}
```

Die Funktion "maximum" wurde hier auf zwei unterschiedliche Datentypen angewendet. Für den Programmierer ist das Verhalten der Funktion auf den Daten entscheidend, die Datentypen sind transparent.Für jeden unterschiedlichen Datentypen muß eine eigene Realisierung der Funktion "maximum" implementiert werden.Für die Funktion "maximum (int a, int b)" gilt die konventionelle Maximumfunktionalität.

```
int    maximum   (int a, int b)          // Int-Int-
```

```
Vergleich
  { if (a > =b)
    return b;
  else
   return a;
  }
```

Das Ergebnis der Relation max(a,b) mit a als int-Array und b als int-Zahl ist lautet
b falls b größer als alle Elemente von a ist, ansonsten das erste Element in a, das größer als b ist.

```
int    maximum   (const int a*, b)          //
Implementierung array und int-Vergleich
  { for(int i=0 ; i < length(a); ++i) {
       if (a [i] > b ) return a[i];
  // Teste Array-Elemente auf a[i] > b
    }
    return b; // b größer als alle Elemente des
//Arrays
  }
```

Beim Überladen von Funktionen muß auf die Auswertungsreihenfolge durch den Compiler geachtet werden, damit sowohl eine korrekte Syntax als auch die beim Programmieren beabsichtigte Semantik eines Programmabschnittes verwendet wird. Daher sollten folgende Regeln beachtet werden:

Wenn sowohl der Identifikationsname einer Funktion als auch die Signatur (Art und Reihenfolge der Parameter) mit anderen Funktionen übereinstimmen, so wird die letzte im Programm aufgelistete Funktionsdefinition verwendet.

Beispiel:

```
void   funktionsname (int *f1,  char ch);
void   funktionsname (int xxx, char zeichen);
```

Wenn der Identifikationsname überladener Funktionen und die Signatur übereinstimmen jedoch der Typ des Rückgabewerts nicht, so stellt dies eine fehlerhafte Zweitdeklaration dar.

Wenn sich Signaturen von zwei oder mehreren Funktionen in bezug auf den Typ oder die Zahl der Argumente unterscheiden, werden diese Funktionen als überladen betrachtet.

Hierbei bestimmt die Reihenfolge der Parameter in der Argumentenliste bereits die neue Funktion. Vorsicht-Ist eine Zuweisung nicht direkt möglich, versucht der Compiler sie über eine Standard-Konversion zu erreichen; gibt es keine passende, versucht der Compiler eine vom Programmierer definierte. Existiert auch diese nicht, wird ein Fehler ausgegeben.

C++ ist bzgl. der Typenprüfung wesentlich genauer als ANSI-C!
Vorsicht bei mehrdeutigen Übereinstimmungen.

Beispiel:

```
void   f1  (unsigned int);
void   f1  (int);
void   f1  (char);
unsigned int var;
void   f1  (var);          // Mehrdeutigkeit!
```

Ein Compiler versucht, Mehrdeutigkeiten aufzulösen. Grundsätzlich sollten diese aber durch den Programmierer vermieden werden. Ein Destruktor darf nicht überladen werden.

Das Überladen von Operatoren:

Neben Funktionen lassen sich in C++ auch Operatoren überladen. Unter Operatoren werden folgende unären und binären Funktionen bzw. Relationen verstanden:

+	-	*	/	%	^	&	I	(	!	=
<	>	+=	-=	*=	/=	%=	I=	<<	>>	<<=
>>=	==	!=	<=	>=	&&	II	++	--	[ ]	()
new	delete	->								

Erklärung:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
^	bitweise XOR
|	bitweise OR
(	bitweise NOT
=	Zuweisungsoperator
<	Kleiner als
>	Größer als
+=	Zuweisung
-=	Zuweisung
*=	Zuweisung
%=	Zuweisung
|=	Zuweisung
<<	Bitverschiebung
>>	Bitverschiebung
<<=	Zuweisung
>>=	Zuweisung
==	logisches Gleich
!=	logisches Ungleich
<=	logisches Kleiner-Gleich
>=	logisches Größer-Gleich
&&	logisches UND
!!	logisches ODER
++	Inkrement (+1)
--	Dekrement (-1)
[ ]	Array-Index
()	Funktionsaufruf, Typkonstruktion

-> Auswahloperator
new, delete dynamische Speicherverwaltung

Der Operator "<<" ist in C++ bereits standardmäßig überladen. Er hat mindestens die folgenden beiden Bedeutungen:

a.)

```
cout << "Test";// Ausgabe eines String auf den
//Bildschirm Der Operator "<<" wird stets zur
//Ausgabe verwendet, wenn cout verwendet wird.
```

b.)

```
unsigned char B1 = 0145;// B1 = 01100101   (in
//Binärdarstellung)
unsigned char Ergeb;

Ergeb = B1 << 1;// Bitweises Verschieben von B1 um
//1 Bit nach links
// Ergeb = 11001010   (in Binärdarstellung)
```

Die Operatoren "+=", "-=", "*=" und "%=" sind Kurzschreibweisen für die folgenden Zuweisungen und für alle einfachen Datentypen (int, float, char, ...) überladen:

a += 1;	ist identisch mit	a = a + 1;
a -= 7.123;	ist identisch mit	a = a - 7.123;
a *= 10;	ist identisch mit	a = a * 10;
b %= 3;	ist identisch mit	b = b % 3;

Es sind logische von binären Operatoren zu unterscheiden! Logische Operatoren geben als Ergebniswert TRUE oder FALSE aus (siehe auch Abschnitt logische Ausdrücke). Binäre (bitweise) Operatoren führen eine logische Funktion auf die angegebenen Parameter aus.

Beispiel:

```
unsigned  char a = 0145;  // Binär  01100101
unsigned  char b;
b = a | a;  // Ergebnis von a OR a an b zuweisen
            // => b = a
if (b == a) // Ist Wert von a identisch mit Wert von
            // b ?
cout << "gleich";// dann Ausgabe des String gleich",
else // ansonsten
  cout << "ungleich";// Ausgabe der Mitteilung
```

```
                        // ungleich".
```

Beispiel:

Im Folgenden wird das Überladen am Beispiel der Konstruktoren der Klasse Bruch erläutert (Bruch.h und Bruch.cpp).

Dateiname: Bruch.h

```
#ifndef _Bruch_h
#define _Bruch_h

#include <stdlib.h>
#include <iostream.h>

// Klasse Bruch

class Bruch
{
protected:  // Schlüsselwort (Erklärung in
            //Abschnitt: Vererbung)
int    zaehler;
int    nenner;

public:// Schlüsselwort (Erklärung in Abschnitt:
       // Vererbung)
Bruch ();       // Default-Konstruktor
Bruch (int);         // Konstruktor aus Zaehler
Bruch (int, int);// Konstruktor aus Zaehler u.Nenner
~Bruch();       // Destruktor

void Ausgabe(); // Ausgabefunktion
Bruch operator * (Bruch); // Multiplikation
                          //mit anderem Bruch
Bruch operator * (int);   // Multiplikation mit
                          //einer ganzen Zahl (int)
```

```
}
#endif // _Bruch_h
```

Dateiname: Bruch.cpp

```
#include "BRUCH.H"

Bruch::Bruch()  // Default-Konstruktor
{
  zaehler =0;   // initialisieren
  nenner = 1;
}

Bruch::Bruch(int z)// Konstruktor mit  Zähler
{
  zaehler =z;   // initialisieren
  nenner = 1;
}

Bruch::Bruch(int z, int n)// Konstruktor mit Zähler
                          // & Nenner
{
  if (n==0)
  {
    cerr << "Fehler: Nenner ist 0! " << endl;
    exit (1);
  };
  zaehler = z;           // initialisieren
  nenner = n;
}

Bruch::~Bruch()               // Destruktor
{
}
```

```
void Bruch::Ausgabe()
{
  cout << zaehler <<"/" << nenner << endl;
}

Bruch Bruch::operator * (Bruch b)
{
  return Bruch (zaehler * b.zaehler, nenner *
b.nenner);
}

Bruch Bruch::operator * (int f)
{
  return Bruch (f * zaehler, nenner);
}
```

Die Konstruktoren der Klasse Bruch unterscheiden sich lediglich durch die Parameter mit denen sie aufgerufen werden. Dadurch entscheidet der Compiler selbst, welcher der Konstruktoren "geeignet" ist.

```
int main ()
{
cout << "Testprogramm für die Klassen Bruch und
Bruch" << endl << endl;

Bruch a;
Bruch b (4);
Bruch c(7,9);
a.Ausgabe();    //Liefert das Ergebnis:  0/1 auf
dem
           //Bildschirm
b.Ausgabe();    //Liefert das Ergebnis:  4/1 auf
dem
           //Bildschirm
c.Ausgabe();    //Liefert das Ergebnis:  7/9 auf
```

```
   dem
                //Bildschirm
}
```

Streams und File I/O

In diesem Abschnitt wird die grundsätzliche Eigenschaft der Ein- und Ausgabe mit Hilfe von Streams vorgestellt. Es handelt sich hierbei um eine Hierarchie voneinander abgeleiteten Klassen. Leider existiert in C++ noch keine einheitlich standardisierte Form der I 10 - Bibliotheken, so daß keine einheitliche Verwendung, z. B. der Namen der header-Dateien möglich ist. Glücklicherweise gibt es jedoch sehr viele Übereinstimmungen unter den bekanntesten Compilern.
Ein Stream ist im Gegensatz zu einem festen Datum als eine unendliche Folge von Daten zu verstehen. Streams sind also "Datenströme", in die Informationen "eingeleitet" werden oder die Daten kontinuierlich liefern.
In C++ werden drei große Klassen von Datenströmen unterschieden:

istream

ostream

iostream

Weitere Streams werden der Dateiverwaltung zugeordnet:

ofstream Ausgabestrom auf eine Datei

ifstream Eingabestrom aus einer Datei

fstream Ein-/Ausgabestrom über Dateien

Die Objekte cout, cin und cerr sind vordefiniert, sobald die Header-Datei iostream.h in das Programm mit `#include` aufgenommen wird. Die Objekte `cin`, `cout`, `cerr` sind lediglich Default-namen. Um Objekte oder Zeichenströme lesen bzw. schreiben zu können, werden die shift-Operatoren << bzw. >> überladen. Diese Operatoren agieren auf den Objekten, die den Datenströmen zugeordnet wurden.

Beispiel:

```
float  f_wert = 1.235;    // Programm gibt sowohl
Double d_wert = 1.222;    // float, double, als
char   * name = "Joern";  // auch strings aus.


cout << f_wert;
cout << d_wert;
cout << name;
```

Da der Operator >> bzw. << als Ergebnis stets eine Referenz auf das stream-Objekt selbst zurückgibt, können die Operatoren auch mehrfach hintereinander ausgeführt werden. Die Leserichtung ist stets von links nach rechts.

Beispiel:

```
cout << f_wert << d_wert << name;//Entspricht obigem
```

Manipulatoren

Manipulatoren sind spezielle Objekte, deren Aufgabe es ist, die Aus- bzw. die Eingabe von Daten zu steuern. Mit der Anwendung eines Manipulator-Objektes ist nicht immer eine Ausgabe oder eine Eingabe verbunden.

Es gibt parameterlose und parameterbehaftete Manipulatorobjekte. Die meisten Objekte werden über die Datei "iostream.h" gelesen. Zusätzlich benötigt wird die Header-Datei "iomanip.h", falls parametarisierte Manipulatoren genutzt werden sollen.
Manipulatoren werden speziell zur Formatierung herangezogen. Die wichtigsten sind:

```
flush       Ausgabepuffer leeren
endl        'In' ausgeben und Ausgabepuffer leeren
ends        'Io' ausgeben und Ausgabepuffer leeren
ws          Trennzeichen überlesen
dcc         Dezimalkonvertierung
hex         Hexadezimalkonvertierung
oct         Oktalkonvertierung
setw(int n) Ausgabebreite setzen für das unmittelbar
folgende Datum
setprecision(int n) Setzen der Ausgabegenauigkeit
(Stellen hinter dem Punkt)
setfill(int c) Setzen des Füllzeichens (default ' ')
```

Parameterlose Manipulatoren sind Funktionen vom Typ

```
ostream & manipulatorname (ostream & str);
```

wobei `ostream` durch die anderen Stream-Klassen ersetzt werden kann.

Format-Funktionen

Ein weiterer Weg, die Formatierung von Streams zu beeinflussen, liegt in der Verwendung von Format-Funktionen. Diese sind der Klasse `ios` zuzuordnen, aus der z. B. `ostream`, `istream`, etc. abgeleitet werden. Daher sind die Einstellungen die durch die ios-Memberfunktionen definiert werden, auch bei den Ein- Ausgabe-Streams wirksam.
Die wichtigsten Funktionen sind:

```
ios::dec         Die Ausgabe soll dezimal
dargestellt werden
ios::hex         hexadezimal
ios::oct         oktal
ios::left        linksbündig
ios::right       rechtsbündig
ios::internal    mit Füllzeichen zwischen Basis und
Wert
ios::showbase    Bei der Ausg. soll zusätzl. die
Basis ausgegeben werden
ios::showpoint   Der Punkt soll nicht unterdrückt
```

```
werden
ios::showpos         Ein 't'-Zeichen soll vor der
Zahl ersch. bei pos. Zahlen
ios::uppercase  Hex-Ausgabe mit Großbuchstaben
(OX1A22F)
ios::scientific Gleitkommadarstellung mit exponent-
Format (1.23e4)
ios::fixed      Gleitkommadarstellung mit fixed-
Format (12.34)
ios::width(n)   entspricht set w (n)
ios::precision(n)entspricht set precision (n)
```

Die in der Tabelle angegenenen ios-Flags werden mit Hilfe der Funktion

```
long ios:: setf (long setBits, long delete Bits =
OL)
```

gesetzt bzw. gelöscht.
Vorsicht:

Der Rückgabewert der Funktion ist eine `long`-Zahl und kein `stream`-Objekt, so daß der Einbau dieses Aufrufs in eine >> bzw. << zur Ausgabe des Funktionswertes führt. Dieser Wert entspricht dem alten Flageintrag bevor `setf` zur Wirkung kommt.

Die einzelnen Bits sollten sinnvoll gesetzt werden, d. h. sie sollten sich nicht widersprechen. Ansonsten werden default-Werte angenommen.

Beispiel:

```
cout.width (17);
cout.setf(ios::scientific, ios::scientific |
                          ios::fixed);
cout.precision (3);
cout.fill ('*');
cout << 317.2299 << endl;

Ergibt als Ausgabe: ********3.172e+02
Gesamte Stellenzahl 17
```

Bemerkungen: Die Breite der Ausgabe wird nicht fest eingehaltene, wenn die Stellenzahl der Ausgabe größer ist als die vorgeschriebene Breite. Somit entspricht diese Zahl eher einer unteren Schranke, die mindestens erfüllt werden muß. Es wird immer bis auf diese Stellenzahl mit Füllzeichen aufgefüllt.
Um die Oder-Anweisung in der `setf`-Funktion zum Löschen der Bits kompakter zu schreiben, gibt es vordefinierte Bit-Masken

```
basefield = dec | oct | hex
adjustfield = left | right | internal
floutfield = scientific | fixed
```

Zur Eingabe von Objektdaten über streams wird standardmäßig die Klasse `istream` verwendet. Die Eingabe erfolgt unter der Verwendung des Operators >>, als Standardeingabestrom wird ein verwendet. Die zuvor besprochenen Formatoperatoren können hier auch eingesetzt werden.
Der Vorteil beim Einsatz des objektorientierten Konzepts liegt nun in der Nutzung der "Eigenintelligenz" des Operators, >>".

Eigenschaften des Operators >>:

Das Ergebnis der Anwendung von cin >> x ist stets eine Referenz auf das Objekt der Klasse `iostream`.

Um den Erfolg bzw. den Status einer Eingabe zu verfolgen, werden Zustandsvariablen gesetzt. Darüber hinaus wurde der Operator "!" auf streams definiert, um (analog zur C-Syntax) den Erfolg einer Streamoperation testen zu können. "!" gibt einen boolschen Wert aus.

Beispiel:

```
if (!(cin>>x))
{
  // Dann Fehler aufgetreten
}
```

Bereits durch die Syntax eines Programms wird der erwartete Typ der Eingabedaten festgelegt.

Beispiel:

```
int x;
float y;
char name [10];
cin >> x >> y >> name;
```

Abhängig von den eingegebenen Daten selbst (falls keine `basefield` Einstellung besetzt ist), wird die Eingabe wie folgt interpretiert:

`octal`	falls erstes Zeichen eine „0“ (Null) und sonst nur Ziffern 0-9
`hexadez.`	falls die ersten Zeichen „0x“, sonst A-F und 0-9
`dez`	falls als erste Zeichen 1, 2,...,9 sonst 0-9

Führende Leerzeichen werden stets überlesen.
Das Lesen eines Operanden endet, wenn:

der Datenstrom mit einem EOF beendet wird

ein Zeichen erkannt wird, welches nicht zum Typ des zu beschreibenden Objektes paßt.

die eingestellte width eines Objektes überschnitten wird.

selbstdefinierte Eigenschaften von Objekten erkannt werden.

ein Fehler (Hardware, etc.) aufgetreten ist

C++ stellt zusätzlich zu den Operatoren >> bzw. << Memberfunktionen der Klassen `istream` bzw. `ostream` zur Verfügung:

```
int istream::get()
```
..extrahiert ein einzelnes Zeichen aus dem Eingabestrom und liefert dieses als Ergebnis zurück, darunter auch das EOF-Zeichen und whitespaces.

```
istream& istream::get(char &c)
```
..liest das nächste Zeichen auf c, whitespaces und EOF werden nicht ignoriert. Das Ergebnis ist der zu bearbeitende `iostream`.

```
ostream& ostream::put(char &c)
```
..schreibt ein Zeichen auf den `ostream` (analog zu getle)).

```
istream& istream::get(char *str, int lng, char trenn= '\n')
```
..liest maximal lng -1 Zeichen auf den String str. Es wird die Eingabe abgebrochen, falls ein gelesenes Zeichen identisch mit "trenn" ist.

```
istream& istream::getline(char *str, int lng, char trenn = '\n')
```
..entspricht dem get-Befehl; '/n=', wird in str. mit aufgenommen.

```
istream& istream::read(char *str, int lng)
```
Diese Funktion liest lng Zeichen in str ein. Der Einlesevorgang kann vorher abgebrochen werden (EOF,etc.)

```
ostream& ostream::write(char *str, int lng)
```
Diese Funktion gibt lng Zeichen aus str aus.

```
istream & istream:: ignore lint anz, char trenn = EOF)
```
Diese Funktion überliest anz Zeichen. Im Falle, daß ein Zeichen identisch mit dem in trenn angegebenen Wert gelesen wurde, kann der Lesevorgang auch vorzeitig (Zahl der gelesenen Zeichen < anz) abgebrochen werden. "trenn" ist default mit EOF belegt.

```
int istream::gcount (void)
```
Diese Funktion gibt als Rückgabewert die Zahl der beim letzten get-, getline -bzw. read - Aufruf gelesenen Zeichen aus.

```
ostream & ostream:: flush (void)
```
..leert den Ausgabepuffer, in dem alle darin befindlichen Zeichen ausgegeben werden.

```
int istream:: peek (void)
```
..liest das nächste Zeichen aus der Eingabe, ohne es dem Datenstrom zu entnehmen. Hiermit wird eine kontextsensitive Verhaltensweise eines Programms erreicht werden. Ein mehrfaches "peek" liefert identische Zeichen.

```
istream & istream:: putback (char c)
```
..schreibt ein Zeichen in den Eingabestrom, so daß als nächstes Zeichen c gelesen wird. Gelesene Zeichen können so wieder in den Eingabestrom zurückgelangen.

Status-Abfragen

Speziell beim Lesen unbekannter Zeichenströme können Fehler und unerwartete Zeichen auftreten. Typische Zeichen sind:

Gelesenes Zeichen paßt nicht zum Datentyp und kann auch nicht konvertiert werden.

Das Datenstromende ist erreicht worden.

Ein Hardware-Fehler ist aufgetreten.

Diese Stream-Eigenschaften werden als Flags in einen Statuswort in `ios` gespeichert. Die Stati von `io_state` sind:

`ios::goodbit`	0x00	kennzeichnet den fehlerfreien Stream
`ios::eofbit`	0x01	kennzeichnet das Erreichen des Endes eines Streams
`ios::failbit`	0x02	letzter Vorgang nicht korrekt abgeschlossen
`ios::badbit`	0x04	Fataler Fehler mit Datenverlust
`ios::hardfail`	0x80	Hardware-Fehler erkannt

Der Unterschied zwischen `failbit` und `badbit` liegt darin, daß bei `failbit` der Datenstrom erhalten werden konnte, ohne daß mit Datenverlusten zu rechnen ist und `badbit` jedoch mit Datenverlust einhergeht. Das Lesen des Datenstroms wird stets unterbrochen.

Beispiel [RRZN]:

```
int wert;
cin >> wert;     // Eingabe: xyz failbit gesetzt, da
            // Zaehler erwartet werden
cin >> wert;     // Eingabe: 0xyz badbit und failbit
            // gesetzt, da x verloren geht. y ist im
            // Puffer und wird als nächstes gelesen.
```

Um den Status eines Stroms zu lesen, gibt es die Anweisungen:

```
int ios::rdstate(void)
```

Um den Status eines Stroms zu löschen oder speziell zu setzen, wurde die Anweisung

```
void ios::clear(int status = 0)
```

definiert.

Beispiel:

```
int Konto::lese_Nummer(istream & strm){
  int Knt_nmr;
  ifl!(strm >> Knt_nmr){
    return -1;
  }
  ifl!(Knt_nmr < = 0){
    strm.clear (strm.rdstate() I ios::failbit);
  } // Fehler im Stream
  return Knt_nmr;
}
```

I/O-Operatoren auf eigenen Datentypen

Der wesentliche Vorteil von Streams liegt darin, daß eigene Ein- und Ausgabemechanismen für selbstdefinierte Objekte geschaffen werden können. Hierzu werden typischerweise die Operatoren >> bzw. << überladen und auf eigene Objekte angewendet.
Vorsicht: Da der linke Operand immer der Stream ist, können << bzw. >> nicht als Memberfunktionen deklariert werden. Sollen diese auf private - Elemente des rechten Operanden zugreifen, müssen sie als `friend` deklariert werden.
Im folgenden soll ein längeres Beispiel zur Verwendung von streams vorgestellt werden. Hierbei sollen Konten mit Kontonummer und Inhabername eingerichtet, beschrieben und ausgelesen werden.

Beispiel:

Headerfile: "Konto.h"

```
class Konto {


public:
int KontoNo;
int Betrag;


int Merke_KontoNo(int Nummer);
int Schreibe_KontoNo();
int Merke_Betrag(int amount);
int Schreibe_Betrag();
};
```

Realisierung: "Konto.c"

```
#include <iostream.h>
#include <Konto.h>


int Konto::Merke_KontoNo(int Nummer){
  KontoNo = Nummer;
  return(1);
}


int Konto::Schreibe_KontoNo()
{
  cout << KontoNo << endl;
  return(1);
}
```

```
int Konto::Merke_Betrag(int amount)
{
  Betrag = amount;
  return(1);
}
int Konto::Schreibe_Betrag()
{
  cout << Betrag << endl;
  return(1);
}
```

Anwendung: "main.c"

```
#include <iostream.h>
#include <Konto.h>

int main()
{
// Deklaration
  Konto kto1;
  Konto kto2;
  Konto kto3;
// Zuweisen von Werten zu einem Objekt
  kto1.Merke_KontoNo(4711);
  kto1.Merke_Betrag(43);
  kto2.Merke_KontoNo(2211);
  kto2.Merke_Betrag(3);
  kto3.Merke_KontoNo(007);
  kto3.Merke_Betrag(456);
// Auslesen der Werte
  kto1.Schreibe_KontoNo();
  kto2.Schreibe_Betrag();
  kto3.Schreibe_KontoNo();
  kto3.Schreibe_Betrag();
```

```
  return(0);
}
```

Ein-/Ausgabe mit Dateien

Die Ein- bzw. Ausgabe von Daten unter Zuhilfenahme von Dateien erfordert das Einbinden der Header-Files iostream.h und fstream.h.

```
# include < iostream.h>
# include < fstream.h>
```

Ein wesentlicher Vorteil der Datei-Stream-Klassen in C++ gegenüber den Mechanismen im herkömmlichen C liegt darin, daß die Dateien bei der Deklaration der Objekte automatisch geöffnet werden, bzw. geschlossen werden wenn die Lebensdauer eines Stream-Objektes erlischt. Jedes mit einer Datei verknüpfte Objekt ist von einer der drei Klassen

ifstream	Eingabestrom
ofstream	Ausgabestrom
fstream	Ein- und Ausgabestrom

Diese Klassen wurden für die Dateibearbeitung von `istream`, `ostream` bzw. `iostream` abgeleitet und verfügen daher über die gleichen Memberfunktionen, einschließlich der überladenen "<<"- bzw. ">>"-Operatoren.

Beispiel:

```
#include <fstream.h>
#include <iostream.h>
#include <iomanip.h>

int main (){    // ASCII - Zuordnungstabelle
  ofstream *outfile = new ofstream ("OutDatei.txt",
                                    ios::out);
  if (! outfile){ // Fehlerbehandlung und Ende
  }
  for(int i = 32; i < 256; i++){
  *outfile  << "Zahl:" << setw(3) << i <<" "
            << "Zeichen:" << (char) i << endl;
  }
  delete outfile;
}
```

Natürlich muß nicht immer ein Objekt mit `new` generiert werden. Ebenfalls ist auch die Angabe des Dateinamens bei der Deklaration nicht sofort notwendig. Eine Datei kann selbstverständlich auch nach der Deklaration geöffnet werden und explizit auch wieder geschlossen werden.

Dies geschieht mit den Anweisungen

```
void fstream::open(const char *dateiname, int modus,
                                          int prot)
void fstream::close(void)
```

Mit "modus" werden verschiedene Arten der Dateibehandlung charakterisiert. Die verschiedenen Modi werden durch ios-Flags beschrieben:

`ios::out`	Öffnen zum Schreiben (default für ofstream)
`ios::app`	Schreiben an das Dateiende und (nur) dahinter.
`ios::ate`	Positionieren auf das Dateiende ("at end"). Schreiben auch vor Dateiende möglich.
`ios::in`	öffnen zum lesen (default für ifstream)
`ios::trunc`	Alten Dateiinhalt löschen. (default, wenn nicht app oder ate angegeben)
`ios::nocreate`	Datei muß existieren
`ios::noreplace`	Datei darf nicht existieren
`ios::binary`	unter DOS zum Unterdrücken von CR/LF. Es wird LF am Zeilenende geschrieben.)

Typischerweise werden die Flags durch "I"- oder gleichzeitig gesetzt, wie z. B. :

```
ofstream datei("xyz.out", ios::out|ios::app|
                                  ios::nocreate);
```

Der wichtigste Unterschied zu den Standard-Strömen (`istream`, `ostream`, `iostream`) liegt bei der Dateibearbeitung im wahlfreien Zugriff auf Daten im Strom. Hierzu kann der Zugriff auf ein Datum durch die Positionierung innerhalb einer Datei gesteuert werden.
Hierzu werden die Anweisungen

```
fstream&fstream::seekg(long pos)
fstream&fstream::seekg(long offset, int bezug)
fstream&fstream::seekg(long pos)
fstream&fstream::seekp(long offset, int bezug)
```

definiert.
`seekg` stellt den Lesezeiger durch Positionieren absolut (mit "pos") oder relativ zu einem vorgegebenen Bezugspunkt "bezug" durch "offset" ein.
`seekp` stellt analog den Schreibzeiger ein.
Beide Funktionen sind auch für die Klassen ifstream und ofstream definiert. Um einfache Bezüge zu kennzeichnen wurden drei Konstanten eingeführt:

`ios::beg`	0x0	Dateibeginn
`ios::cur`	0x1	relativ zur aktuellen Position
`ios::end`	0x2	Dateiende

Zur Ermittlung der aktuellen Position dienen:

```
long fstream::tellg (void)
long fstream::tellp (void)
```

Unter DOS sollte im Textmodus stets tellg / tellp gemeinsam mit seekg / seekp Befehlen genutzt werden, da die Position in der Datei durch die Kombination CRLF zur Zeilenendenkennzeichnung "verfälscht" wird.

Beispiel:

```
fstream datei("hallo.in", ios::nocreate|ios::ate|
                                  ios::binary);
```

```
if(!datei){
  cerr << "Fehler!"; return -1;}// Fehler

datei.seekg(0, ios::geb);      // Auf den
                               // Dateianfang
datei.read (obj, size of (obj));// Objekt "obj"
                               // lesen
datei.seekp (0, ios::end);     // An das Ende der
Datei
datei.write (obj, size of (obj));// "obj" schreiben
datei.close;                        // Datei
schließen
```

Vererbung

Das wichtigste Merkmal der Vererbung besteht darin, daß Eigenschaften einer Klasse von einer anderen Klasse abgeleitet werden können. Für neue Klassen müssen nur noch Neuerungen implementiert werden! Eigenschaften, die schon einmal in einer anderen Klasse implementiert wurden, können übernommen und müssen nicht noch einmal implementiert werden. Dies führt neben einer Einsparung an Code zu einem Konsistenzvorteil, da gemeinsame Eigenschaften nicht an mehreren Stellen stehen und somit nur einmal geprüft oder geändert werden müssen. Die Klasse, von der geerbt wird, nennt man Basisklasse. Die Klasse, die erbt, nennt man abgeleitete Klasse.

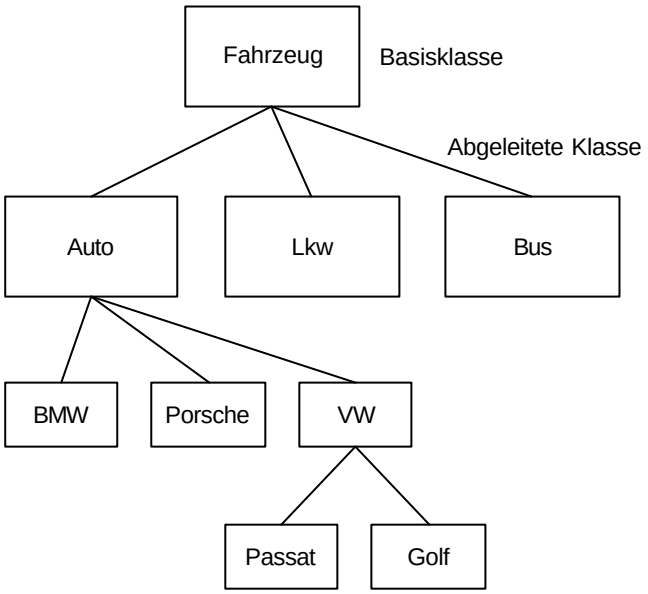


Bild 0.1: Vererbung

Da eine abgeleitete Klasse selbst Basisklasse sein kann, kann eine ganze Klassenhierarchie entstehen (s. o.). Die Klasse Fahrzeug beschreibt Eigenschaften von Fahrzeugen. Die Klasse Auto erbt diese Eigenschaften und erweitert sie. Die Klasse BMW erbt die Eigenschaften von Auto (also die erweiterten Eigenschaften von Fahrzeug) und erweitert sie weiter.

C++ unterstützt Mehrfachvererbung (eine abgeleitete Klasse kann mehrere Basisklassen besitzen).

Konstruktoren werden nicht an die abgeleitete Klasse vererbt.

Bei der Deklaration der Konstruktoren werden die Parameter durchgereicht.

Zugriffsrechte:

Schlüsselwörter:

private: Auf diese Komponenten kann nur diese Klasse zugreifen.

protected: Die eigene und die abgeleiteten Klassen haben Zugriff.

public: Diese Komponenten sind öffentlich.

Zur Erläuterung wird die Klasse "KBruch" (kürzbarer Bruch) eingeführt.

Beispiel:

Dateiname: KBruch.h

```
ifndef _KBruch_h
#define _KBruch_h
#include "BRUCH.H"        // header-Datei der
Basisklasse

class KBruch : public Bruch
{
protected:
bool kuerzbar;       // true: Bruch ist kuerzbar
unsigned ggt() const;// Hilfsfunktion größter
                     //gemeinsamer teiler
public:
KBruch (int , int);       // Konstruktor
void kuerzen();           // Bruch kuerzen
bool istkuerzbar() const; // liefert ob kuerzbar
};
#endif // _KBruch_h
```

Dateiname: KBruch.cpp

```
#include "KBRUCH.H"

inline static int min (int a, int b) //Hilfsfunktion
{ return a < b ? a: b;    }

KBruch::KBruch (int z = 0, int n=1) : Bruch(z,n)
{ kuerzbar = (ggt() > 1);}

unsigned KBruch::ggt() const  // Ermittelt den
                     //größten gemeinsammen Teiler
{
```

```
    unsigned teiler = min(abs(zaehler), nenner);
    while (zaehler % teiler !=0 || nenner % teiler!=0)
    teiler--;
    return (teiler);
  }

  bool KBruch::istkuerzbar() const
  { return (kuerzbar);}

  void KBruch::kuerzen()    // Der Bruch wird
                 //gekürzt, wenn er kürzbar ist.
  {
    if (kuerzbar)          // falls kuerzbar
    {
      int teiler = ggt();

      zaehler /= teiler;    // teilen
      nenner /= teiler;
      kuerzbar = false;     // nicht mehr kuerzbar
    };
  }
```

Zunächst muß die Header-Datei der Basisklasse eingebunden werden, da die Deklaration der abgeleiteten Klasse darauf aufbaut:

```
#include "BRUCH.H"
```

Entscheidend für die Tatsache, daß es sich um eine abgeleitete Klasse handelt, ist dann die eigentliche Klassendeklaration:

```
class KBruch : public Bruch
{
.
.
.
};
```

Eine abgeleitete Klasse wird dadurch deklariert, daß ihre Basisklasse hinter einem Doppelpunkt und einem optionalen Zugriffsschlüsselwort angegeben wird. Dabei spielt es keine Rolle, ob die Basisklasse selbst eine abgeleitete Klasse ist. Das optionale Zugriffschlüsselwort gibt an, ob und inwiefern der Zugriff auf geerbte Komponenten weiter eingeschränkt wird:

public: es gibt keine weiteren Einschränkungen

protected: alle public-Komponenten der Basisklasse sind ab der aktuellen Klasse nur noch protected (nur noch für die eigene und die abgeleiteten Klassen sichtbar)

private: alle Komponenten der Basisklasse werden zu privaten Komponenten

Beispiel:

```
int main ()
{
KBruch x(99,12);

x.Ausgabe();    //Liefert das Ergebnis: 99/12 auf
dem
           //Bildschirm
cout << "x ist " << (x.istkuerzbar() ? "kuerzbar !"
: "unkuerzbar!") << endl;
cout << "kuerzen!" << endl;
x.kuerzen();

cout << endl << "x = ";
x.Ausgabe();    //Liefert das Ergebnis: 33/4 auf
dem
           //Bildschirm
};
```

Literatur zur Programmiersprache C++

[10] Joussutis, N. *Objektorientierte Programmierung in C++*
Addison-Wesley, 1994

[11] Stroustrup, B. *Die C++ Programmiersprache*
Addison-Wesley, 1992

[12] Stroustrup, B. *The Design and Evolution of C++*
Addison-Wesley, 1984

[13] Lippman, S. *C++-Einführung und Leitfaden*
Addison-Wesley, 1991

[14] Booch, G. *Object-oriented Analysis and Design*
Standardwerk zu objektorientierter Analyse und Design

[15] Kruglinski, D. *Inside Visual C++ 4.2*
Microsoft Press